III. Hierarchical Program Structures

OLE-JOHAN DAHL AND C. A. R. HOARE

1. INTRODUCTION

In this monograph we shall explore certain ways of program structuring and point out their relationship to concept modelling.

We shall make use of the programming language SIMULA 67 with particular emphasis on structuring mechanisms. SIMULA 67 is based on ALGOL 60 and contains a slightly restricted and modified version of ALGOL 60 as a subset. Additional language features are motivated and explained informally when introduced. The student should have a good knowledge of ALGOL 60 and preferably be acquainted with list processing techniques.

For a full exposition of the SIMULA language we refer to the "Simula 67 Common Base Language" [2]. Some of the linguistic mechanisms introduced in the monograph are currently outside the "Common Base"*.

The monograph is an extension and reworking of a series of lectures given by Dahl at the NATO Summer School on Programming, Marktoberdorf 1970. Some of the added material is based on programming examples that have occurred elsewhere [3, 4, 5].

2. PRELIMINARIES

2.1 BASIC CONCEPTS

Our subject matter as programmers is a special class of dynamic system, which we call computing processes or data processes. A programming

* The Simula 67 language was originally designed at the Norwegian Computing Center, Oslo. The Common Base defines those language features which are common to all implementations. The Common Base is continually being maintained and revised by the "Simula Standards Group", each of whose members represents an organisation responsible for an implementation. 8 organisations are currently represented on the SSG. (Summer 1971).

language provides us with basic concepts and composition rules for constructing and analysing computing processes.

The following are some of the basic concepts provided by ALGOL 60.

(1) A *type* is a class of *values*. Associated with each type there are a number of operations which apply to such values, e.g. arithmetic operations and relations for values of type **integer**.

(2) A *variable* is a class of values of a given type ordered in a time sequence. The associated operations are accessing and assigning its current value. Both can be understood as *copying* operations.

(3) An *array* is a class of variables ordered in a spatial pattern. Associated is the operation of *subscripting*.

Notice that each of the concepts includes a data structure as well as one or more associated operations.

As another example consider machine level programming. The fundamental data structure is a bit string, which is not itself a very meaningful thing. However, combined with an appropriate sensing mechanism it has the significance of a sequence of Boolean values. In connection with a binary adder the bit string has the meaning of a number in some range, each bit being a digit in the base two number system. An output channel coupled to a line printer turns the bit string into a sequence of characters, and so forth. Thus the meaning of the data structure critically depends on the kind of operations associated with it.

On the other hand, no data process is conceivable which does not involve some data. In short, data and operations on data seem to be so closely connected in our minds, that it takes elements of both kinds to make up any concept useful for understanding computing processes.

2.2. HIGHER LEVEL CONCEPTS

As the result of the large capacity of computing instruments, we have to deal with computing processes of such complexity that they can hardly be constructed and understood in terms of basic general purpose concepts. The limit is set by the nature of our own intellect: precise thinking is possible only in terms of a *small* number of elements at a time.

The only efficient way to deal with complicated systems is in a hierarchical fashion. The dynamic system is constructed and understood in terms of high level concepts, which are in turn constructed and understood in terms of lower level concepts, and so forth. This must be reflected in the *structure* of the program which defines the dynamic system; in some way or another the higher level concepts will correspond to program components.

The construction of concepts suitable in a given situation is a creative process which often requires insights obtained at later stages of the system construction. Therefore, as programmers are painfully aware, any software project tends to be a complicated iterative process involving reconstruction and revision at each stage.

Each concept necessarily concerns a limited aspect of the system and should correspond to a piece of program obtained by *decomposition* of the total program. Good decomposition means that each component may be programmed independently and revised with no, or reasonably few, implications for the rest of the system. Thereby the total iteration process may be speeded up.

Any useful concept has some degree of generality, i.e. it is a *class* of specialised instances. In other words one tries to group phenomena occurring in a dynamic system into *classes* of phenomena and to describe each class by a single piece of program.

As an obvious example consider the arithmetic operations involved in a matrix multiplication. They may all be classified as dynamic instances (executions) of the single statement

$$C[i, j] := C[i, j] + A[i, k] \times B[k, j];$$

provided that the matrix coefficients are classified as elements of two-dimensional arrays A, B, and C, and that the variables i, j, and k are given values according to a certain pattern.

The above statement is not sufficiently well decomposed to be thought of as a "concept". The procedure declaration below, however, defines in a concise way the concept of matrix multiplication.

It is important that a concept may be classified as a *syntactic category* (e.g. ⟨block⟩, ⟨procedure⟩) in a general language framework. Structured thought in terms of given concepts implies the construction of sentences, where the concepts have syntactic and semantic relationships to one another. The procedure below is related to other program components through calling sequences (procedure statements).

```
procedure matmult (A, B, C, m, n, p);
    array A, B, C; integer m, n, p;
begin integer i, j, k;
        for i := 1 step 1 until m do
        for j := 1 step 1 until n do
        begin C[i, j] := 0;
            for k := 1 step 1 until p do
            C[i, j] := C[i, j] + A[i, k] × B[k, j]
        end
end;
```

The parameter mechanism of procedures in SIMULA deviates somewhat from that of ALGOL 60. The default transmission mode is by value for ordinary simple ⟨type⟩ parameters, and by "reference" for parameters of other kinds. This deviation is introduced for various pragmatic reasons, one of them being the compatibility with class declarations (cf. 3.1). Thus, in the above procedure the parameters *i*, *j*, and *k* are called by value, *A*, *B*, and *C* by reference.

2.3. BLOCKS AND BLOCK INSTANCES

One of the most powerful mechanisms for program structuring in ALGOL 60 is the block and procedure concept. It has the following useful properties from the standpoint of concept modelling.

(1) *Duality.* A block head and block tail together define an entity which *has properties* and *performs actions.* Furthermore the properties may include a data structure as well as associated operators (local procedures).

(2) *Decomposition.* A block where only local quantities are referenced is a completely selfcontained program component, which will function as specified in any context. Through a procedure heading a block (procedure) instance may interact with a calling sequence. Procedures which reference or change non-local variables represent a partial decomposition of the total task, which is useful for direct interaction with the program environment.

(3) *Class of instances.* In ALGOL 60 a sharp distinction is made between a block, which is a piece of program text, and a dynamic block instance, which is (a component of) a computing process. An immediate and useful consequence is that a block may be identified with the *class* of its potential activations. (Strictly speaking a "block" in this context means either the outermost block or a block immediately enclosed by a dynamic block instance.) Through the recursion mechanism of ALGOL 60 different instances of the same block may co-exist in a computing process at the same time.

(4) *Language element.* A block is itself a statement, which is a syntactic category of the language. Furthermore, through the procedure mechanism, reference to a block may be dissociated from its defining text.

Referring back to our earlier discussion it appears that the ALGOL block mechanism has all the properties required of a concept modelling mechanism. On closer inspection, however, it turns out that the composition rules and interaction mechanisms provided place certain restrictions on the range of concepts to be formulated.

In ALGOL 60, the rules of the language have been carefully designed to ensure that the lifetimes of block instances are nested, in the sense that those instances that are latest activated are the first to go out of existence. It is this feature that permits an ALGOL 60 implementation to take advantage of a

stack as a method of dynamic storage allocation and relinquishment. But it has the disadvantage that a program which creates a new block instance can never interact with it as an object which exists and has attributes, since it has disappeared by the time the calling program regains control. Thus the calling program can observe only the results of the actions of the procedures it calls. Consequently, the operational aspects of a block are overemphasised; and algorithms (for example, matrix multiplication) are the only concepts that can be modelled.

In SIMULA 67, a block instance is permitted to outlive its calling statement, and to remain in existence for as long as the program needs to refer to it. It may even outlive the block instance that called it into existence. As a consequence, it is no longer possible to administer storage allocation as a simple stack; a general garbage-collector, including a scan-mark operation, is required to detect and reclaim those areas of store (local workspace of block instances) which can no longer be referenced by the running program. The reason for accepting this extra complexity is that it permits a wider range of concepts to be conveniently expressed. In particular, it clarifies the relationship between data and the operations which may be performed upon it, in a way which is awkward or impossible in ALGOL 60.

3. Object Classes

A procedure which is capable of giving rise to block instances which survive its call will be known as a *class*; and the instances will be known as *objects* of that class. A class may be declared, with or without parameters, in exactly the same way as a procedure:

⟨class declaration⟩::= **class** ⟨class identifier⟩
⟨formal parameter part⟩; ⟨specification part⟩;
⟨class body⟩

⟨class body⟩::= ⟨statement⟩

Any variables or procedures declared local to the class body are called *attributes* of that class; and so are the formal parameters, whether called by value or called by reference. If the class body is not a block, it is regarded as if it were surrounded by block brackets **begin**...**end**.

A call of a class generates a new object of that class. The initial values of those of its attributes corresponding to formal parameters are specified in the actual parameter part of the generator. A generator always appears as a function designator, returning as its value a *reference* to the newly generated object:

⟨object generator⟩::= **new** ⟨class identifier⟩
⟨actual parameter part⟩

In order to be able to refer again to a generated object, it is necessary to store the reference to it in a variable. Variables used for this purpose should be declared as of *reference* type; and the declaration should also be *qualified* by stating the class of objects to which that variable will refer.

⟨reference variable declaration⟩::=
ref (⟨qualification⟩) ⟨identifier list⟩
⟨qualification⟩::= ⟨class identifier⟩

The notation **ref** (⟨qualification⟩) may also be used to declare reference arrays, procedures, and parameters. An analogous mechanism for "record handling" was first proposed by Hoare [6].

There is a neutral reference value **none** which does not refer to any object; and this is automatically assigned as initial value to every reference variable.

Reference values may be assigned, and tested for equality or inequality; but in SIMULA these operations are given special symbols, in order to emphasise the fact that they operate on references to objects, and not upon the current values contained in those objects.

Thus:

:− denotes reference assignment
== denotes reference equality
=/= denotes reference inequality.

Reference values may also be passed as parameters, and they may be returned as the result of a function designator. A special example of such a function designator is of course the object generator which brings the object into existence, and passes back a reference to it as result.

Example:

class *C*(...); ... class body for *C*...;
ref (*C*)*X*;
....
if *X* == **none then** *X*:− **new** *C*(...);

The attributes of any object may be inspected or changed by the technique of *remote identification*. If *X* is a reference variable qualified by class *C*, and *A* is an attribute identifier (i.e. local quantity) of that class, then *X.A* refers to the attribute *A* of the object currently referenced by *X*. If *X* has the value **none**, the remote access is erroneous. If *A* is a variable attribute, *X.A* may appear to the left of an assignment, as an actual parameter, or in an expression. If *A* is a procedure attribute, *X.A* may appear as an actual parameter, or as a procedure statement or function designator, in which case it will be immediately followed by an actual parameter part. In short, a remote identifier *X.A* may appear in any context in which an ordinary identifier may appear, except for a defining occurrence in a declaration.

In addition to reference variables, every reference parameter, function or expression has a qualification associated with it. In every assignment to a reference variable, it is possible to check that the assignment is valid, by comparing the qualifications of the left hand and right hand sides. SIMULA 67 has been designed to ensure that this check can be carried out wholly at compile time, thus avoiding the inefficiency of run-time checking, and the inconvenience of run-time error. Furthermore all remote identifiers can be checked at compile time to ensure that the combination of reference variable and attribute identifier is valid, so that the only error that has to be detected at run-time is when the reference variable has the value **none**.

The following sections provide examples of concepts modelled by means of class declarations.

3.1. FREQUENCY HISTOGRAM

A frequency histogram of a real random variable with respect to given disjoint intervals can be represented by a table of integers $T_0, T_1, \ldots, T_n$, where T_i is the number of observations falling in the ith interval. A sequence of increasing numbers $X_1, X_2, \ldots, X_n$ partitions the real axis into the following $n + 1$ intervals:

$$\langle -\infty, X_1\rangle, (X_1, X_2\rangle, \ldots, (X_n, \infty\rangle.$$

The ith relative frequency ($i = 0, 1, \ldots, n$) is equal to T_i/N, where N is the total number of observations tabulated in the histogram.

We wish to represent the concept of a histogram as a self-contained piece of program, which can be incorporated in any subsequently written program which requires it. In a realistic program, it will be necessary to maintain several histograms to tabulate different random variables; for example, it may be necessary to record not only random lengths, but also random weights and random heights, and this will require three separate histograms, existing simultaneously with each other and with the main program which has generated them and which is using them. Furthermore, the numbers of the intervals and the partitioning values between them may be different in each case. This suggests that the histogram should be declared as a *class*, with two parameters:

```
class histogram (X, n); array X; integer n;
```

where X is a real array of n elements specifying the boundaries of the partitions. The main program will use this class in the following way:

```
begin ref (histogram) heights, weights, lengths;
    real array A[1:7], B[1:12];
    ...initialise A, B...;
    heights:- new histogram (A, 7);
```

```
        weights: – new histogram (B, 12);
        lengths: – new histogram (A, 7);
        ....rest of program....
    end
```

In the rest of the program, the three histograms may be referred to by the names of the three reference variables. In order to record each new observation (say h or w) in the appropriate histogram, the program will contain the corresponding calls on a procedure tabulate:

```
weights.tabulate (w);
heights.tabulate (h);
```

The procedure "tabulate" must therefore be an attribute of the histogram class. Another attribute of the class must be the array T which counts the number of observations in each interval; and also a variable N to count the total number of observations recorded so far. Finally, a function frequency (i) is required so that the relative frequency of observations in the ith interval may be read out. The only action required of the class body is to initialise these variables.

The declaration of the histogram class may be given:

```
class histogram (X, n); array X; integer n;
  begin integer N; integer array T[0:n];
    procedure tabulate (Y); real Y;
      begin integer i; i:= 0;
        while (if i < n then Y < X[i + 1] else false)
          do i:= i + 1;
        T[i]:= T[i] + 1; N:= N + 1
      end of tabulate;
    real procedure frequency (i); integer i;
      frequency:= T[i]/N;
    integer i;
    for i:= 0 step 1 until n do T[i]:= 0; N:= 0
  end of histogram;
```

Note. (1) In SIMULA 67, all simple parameters of a class or a procedure are called by value, even if the value parts are omitted. Arrays and other parameters are called by name.

(2) In SIMULA 67 all variables are automatically initialised on declaration to neutral values, **false** for Booleans, 0 for numbers, **none** for references. Thus in the examples given above the statements $i:= 0$, $N:= 0$, and the loop initialising T could have been omitted.

It seems reasonable to claim that this piece of program adequately represents the concept of a histogram, in that it expresses the close relationship between the data items X, n, T and N, and the operation of tabulation which is to be performed on them. It would be possible, of course, to write the operation in ALGOL 60 as a separate procedure with many parameters:

procedure tabulate (X, n, T, N, y);

which records observation y in the histogram T in accordance with partitions defined by X, and also increments N. But this would be an artificial separation of the operational aspect of the histogram from the data storage aspect; and the failure in adequately representing the concept is evidenced by the complexity of the specification of the procedure and the awkwardness of its use.

It is worth while to explain the effect of creating a new object of class histogram by means of the statement

weights: – **new** histogram (B, 12).

First, a new object is created, consisting of the variables brought into existence by execution of the declarations for T, N, i, and the parameters X and n, which are initialised to B and 12 respectively. The body of the class declaration is now executed to initialise the other variables. On exit from the body, the variables are *not* deallocated. Rather a reference (pointer, address) to them is passed back and assigned to the variable "weights". It is convenient to think of an object as a complete textual copy of the class body (including the specification part), in which the parameters and local variables and arrays correspond to actual storage locations. Thus an object may well contain local procedure (and even class –) declarations, as well as executable statements.

Subsequently, on execution of the procedure call weights.tabulate (w), it is the tabulate procedure local to the object referenced by "weights" that is actually executed, and causes updating of the local attributes T and N of that object and no other.

3.2. GAUSS-INTEGRATION

A definite integral may be approximated by an "n-point Gauss formula", which is a weighted sum of n function values computed at certain points in the integration interval.

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The weights and abscissa values are chosen such as to give an exact result for the integral of any polynomial of degree less than $2n$. By a suitable transformation we find

$$w_i = (b - a)W_i \text{ and } x_i = a + (b - a)X_i,$$

where W_i and $X_i (i = 1, 2, \ldots, n)$ only depend on n, and not on a or b. The idea of Gauss-integration is expressed in the following partly informal class declaration.

```
class Gauss (n); integer n;
begin array W, X[1:n];
    real procedure integral (f, a, b);
                real procedure f; real a, b;
        integral:= Σ(i=1,n) W[i]×f(a + (b − a)×X[i])×(b − a);
    compute W[1], ..., W[n], X[1], ..., X[n] as
    functions of n
end of Gauss;
.....
ref (Gauss) G5, G7;
.....
G5:− new Gauss (5); G7:− new Gauss (7);
.....
...  G5.integral (F, A, B).....G7.integral (F, A, B).....
```

Comments. The variables G5 and G7 refer to the concepts "5-point" and "7-point Gauss-integration". Each of them is a *specialised instance* of the more general concept of "*n*-point Gauss-integration", represented by the class.

A Gauss object computes once and for all the values of its local array elements, after which control returns to the ⟨object generator⟩. The procedure "integral" is intended for repeated use from outside the object.

The example indicates that the *own*-concept of ALGOL is superfluous in this framework.

4. Coroutines

In ALGOL 60, a most powerful method of combining two pieces of program to accomplish some task is to declare one of them as a procedure, and to invoke it (possibly repeatedly) from within the other. However, in some cases the relationship between the two pieces of program is not fairly represented by this form of master/subordinate relationship; and it is better to regard them as *coroutines* operating in some sense at the same level.

A simple example of coroutine structuring is provided by a games-playing program, which calculates its own move and outputs it to its opponent, inputs the opponent's response, computes its next move, and so on until the game is complete. Suppose now that two different programs have been constructed to play the same game, and it is desired to see which of them is the stronger player. The complete program to play the game is very naturally structured from its two component players, but the structuring method is that of the coroutine rather than the subroutine.

Another example of coroutine structuring is provided in a two-pass compiler for a programming language. The first pass normally outputs a long sequence of messages which are subsequently input by the second pass. However, if sufficient main storage is available to accommodate the program for both passes simultaneously, it is possible to arrange for the whole translation to be carried out apparently in a single pass, where the sequence of messages is transmitted *piecewise* from the first pass to the second pass. First, the second pass is executed until it reaches its first request for an input message. The first pass program is then executed until it produces its first output message. The message is then handed over to the second pass, and the process is repeated until the second pass is complete. In some circumstances it might be possible to restructure one of the passes as a subroutine to the other; but since the choice would be arbitrary, it is better to regard the two programs as coroutines.

This case may be distinguished from the games-playing example in that the flow of information is in one direction only, from the first pass program which "produces" it to the second pass program which "consumes" it. This suggests that a single coroutine may profitably be regarded as a complete selfcontained program whose input and output instructions have been replaced by calls upon other coroutines to produce and consume the data. Each time a coroutine passes control to another coroutine for this purpose, it will expect to resume at the next following instruction. The instruction which causes transfer of control to another coroutine is known as

$$\text{resume } (X)$$

where X refers to the coroutine being resumed.

In SIMULA, a coroutine is represented by an object of some class, cooperating by means of resume instructions with objects of the same or another class, which are named by means of reference variables. The communication of information may be accomplished in variables either global to all the objects or local to one of them; a producing coroutine assigns values to these variables, and the consuming coroutine accesses them. In the case of two-way communication, both coroutines may update the same global variables in turn.

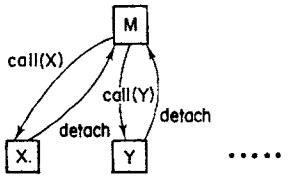
When an object is first generated, it has a subordinate, procedurelike relationship to the block instance which generated it. This is evidenced by the fact that control automatically returns to the generator upon passage through the end of the object. The object does not in general know the identity of its generating block instance; it cannot therefore use a resume instruction to achieve the effect of a coroutine exit. A special, parameterless "detach" instruction is therefore provided by which a generated object can return control to the generator. The generator may then resume the detached object at the point following its (most recently executed) detach instruction by the statement

call (X)

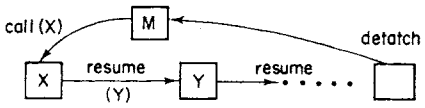
where X is a reference to the detached object. Now the object is again in a subordinate position, with respect to the caller, and has an obligation to return to it either by a detach instruction or by going through its own **end**.

Thus a main program may establish a coroutine relationship with an object that it has generated, using the call/detach mechanism instead of the more symmetric resume/resume mechanism. In this case, the generated object remains subordinate to the main program, and for this reason is sometimes known as a *semicoroutine*. But a semicoroutine may also be a full coroutine with respect to a group of other generated objects, with which it communicates by means of resume statements. In this case, if any of the group issues a detach, control returns to the master program which originally called a particular member of the group. Thus a coroutine issuing a resume statement imposes on the resumed coroutine its own responsibility, eventually to pass control back to the original caller by means of a detach.

Let X and Y be objects, generated by a "master program" M. Assume that M issues a call (X), thereby invoking an "active phase" of X, terminated by a detach operation in X; and then issues a call (Y), and so forth. In this way M may act as a "supervisor" sequencing a pattern of active phases of X, Y, and other objects. Each object is a "slave", which responds with an active phase each time it is called for, whereas M has the responsibility to define the large scale pattern of the entire computation.



Alternatively the decision making may be "decentralised", allowing an object itself to determine its dynamic successor by a resume operation.



The operation resume (Y), executed by X, combines an exit out of X (by detach) and a subsequent call (Y), thereby bypassing M. Obligation to return to M is transferred to Y.

The history of a typical coroutine object may be summarised as follows:

(1) Upon generation, an object starts performing the operations of its class body, and is said to be *operating* and *attached* to (the block instance containing) the object generator which calls it into existence.

(2) The object issues a *detach* statement which returns control to the point at which the object was generated. The object is then said to be detached, but not yet terminated. The detach statement leaves a mark in the body of the object specifying where its operations will be continued. This mark is positioned at the end of the detach statement most recently executed by that object.

(3) Control returns to the object on execution of either a call statement or a resume statement specifying that object by means of its reference parameter. It is then *reattached* to the *calling* block instance if called, or to the original caller if resumed. The object may then temporarily relinquish control again, either by a detach or by a resume, in which case it becomes detached again.

(4) Alternatively, it may relinquish control finally by passing through its **end**, which has the same effect as a detach. But in this case it is said to be *terminated*, and it may not be reactivated either by a call or a resume. However, it remains in existence as an item of data, which may be referenced by remote identification of its attributes, including procedure and function attributes, as in the case of the histogram.

Note. The detach operation represents a coroutine exit out of an *object*, and is only available textually within objects, i.e. textually within class bodies. If issued in a subblock or in a procedure body, a detach instruction still represents an exit out of the (smallest) textually enclosing object. The same is true for the resume instruction (which includes a coroutine exit). The call instruction is, however, available at any point in a program.

4.1. TEXT TRANSFORMATION

As an example of the cooperation of coroutines we take a problem posed by Conway [7]. A text is to be read from cards and listed on a line printer. The cards each contain 80 characters, but the line printer prints 125 characters on each line. It is intended to pack as many characters as possible on each output line, marking the transition from one card to the next only by insertion of an extra space. In the text, any consecutive pair of asterisks is to be replaced by "↑". The end of the text is marked by a special character known as "end".

We assume the existence of a coroutine "incard", which on each resumption will fill the array $C[1:80]$ with characters read from the next card in the card hopper, and pass the card through to the stacker. Also, we are given a coroutine "lineout", which on each resumption will print on the next line of paper the characters from the array $L[1:125]$, and then throw the line.

The task is carried out by three coroutines, which will be known by reference as:

```
ref disassembler, squasher, assembler;
```

The disassembler inputs a card (through C) and outputs individual characters (through $c1$) to the squasher, after inserting a space between cards. The squasher performs the transformation on double asterisks, and outputs individual characters through $c2$ to the assembler. The assembler groups the characters into lines and outputs them; it also detects the "end" character and takes appropriate action.

The required class declarations are:

```
class pass 1;
begin detach;
    while true do
      begin integer i; resume (incard);
          for i := 1 step 1 until 80 do
              begin c1 := C[i]; resume (squasher) end;
          c1 := blank; resume (squasher)
      end infinite loop;
end pass 1;
class pass 2;
begin detach;
    while true do
      begin if c1 = "*" then
```

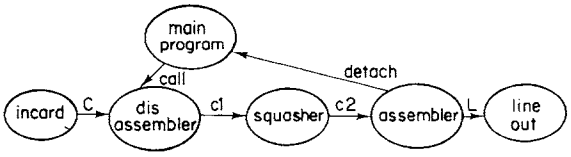
```
            begin resume (disassembler);
                if c1 = "*" then c2: = "↑"
              else begin c2: = "*"; resume (assembler);
                         c2: = c1
                   end;
            end
                                else c2: = c1;
          resume (assembler); resume (disassembler)
        end infinite loop;
end pass 2;
class pass 3;
begin detach;
      while true do
        begin integer i;
            for i: = 1 step 1 until 125 do
                  begin L[i]: = c2;
                        if c2 = "end" then
                        begin for i: = i + 1 step 1 until 125 do
                                  L[i]: = blank;
                              resume (lineout);
                         detach; comment back to main program;
                        end
                                  else resume (squasher)
                  end of this line;
            resume (lineout)
        end infinite loop
end pass 3;
```

The main program generates one instance of each of the passes. Each pass immediately detaches itself from the main program. The system of coroutines is initiated by calling the disassembler. On detection of the end of the task, the assembler issues a detach instruction. Since the assembler obtained control (indirectly) by resume instructions from the disassembler, its detach has the same effect as it would have had if issued by the disassembler, and takes control back to the main program, which then immediately terminates.

The main program is:

```
begin disassembler: – new pass 1;
                squasher: – new pass 2;
               assembler: – new pass 3;
                  call (disassembler);
end
```

The relationships between the five coroutines and the main program may be represented pictorially:



The horizontal arrows represent resume/resume relations. Their direction corresponds to the flow of information; and they are annotated by the name of the variable used to hold the communicated information.

In this example, it is intended that each class should only ever have one object in it; and therefore the full class/generation/reference mechanism is unnecessarily elaborate. The elaboration is inconvenient in that separate names have to be invented for the class and its unique object (e.g. pass 1 and disassembler). Furthermore, in the implementation it should be possible to take advantage of this special case to save both space and time. But SIMULA 67 provides no means of achieving this.

4.2. PERMUTATION GENERATOR

We wish to define a class "permuter" representing the concept of permutations. An object of this class should be capable of generating all permutations of the integers between 1 and n, where n is a parameter of the class. One of the attributes of the class will be an **integer array** $p[1:n]$, which is to be initialised to the value $(1, 2, \ldots, n)$ (representing the identity permutation) when an object of the class is generated. Every subsequent call of the object causes the array p to take a new permutation as value. When all permutations are exhausted, an attribute

Boolean more;

(initially true) will be assigned the value **false**, and the object will terminate.

A typical structure for a program which wishes to inspect all permutations of N numbers will be:

```
ref (permuter)P;
P: - new permuter (N);
while P.more do
    begin...inspect P.p...; call (P) end;
```

The structure of the permuter class will be a semicoroutine, which issues a detach instruction after each updating of p:

```
class permuter (n); integer n;
    begin integer array p[1:n];
          Boolean more;
          integer q;
          for q:= 1 step 1 until n do p[q]:= q;
          more:= true;
        ...generate all permutations of p,
          issuing a "detach" after each of them...;
          more:= false
    end
```

It remains to find an algorithm to carry out all the permutations of $p[1], p[2], \ldots, p[n]$, and restore them to their original state. This algorithm may be recursively structured. Let us assume that we know how to generate *all* permutations of the numbers

$$p[1], p[2], \ldots, p[k-1],$$

and finally return these to their original state. This will be accomplished by a procedure call

permute $(k-1)$.

Now all that need be done is to use this procedure to permute every *combination* of $k-1$ numbers from the original k numbers. Thus there must be k calls of permute $(k-1)$, and on each call, exactly one of the $p[i]$ for $1 \leqslant i \leqslant k$ must be excluded from the operation. A good way of excluding it is to exchange its value with that of $p[k]$, which remains untouched by permute $(k-1)$. In order to ensure that each of the k values is excluded exactly once, we may take advantage of the assumption that the procedure returns the given sequence unchanged. In that case $p[k]$ will be assigned each value once

if we first swap $p[1]$ and $p[k]$, then $p[2]$ and $p[k]$, ..., and then $p[k-1]$ and $p[k]$. Thus we are led to the following kernel:

```
integer i;
permute (k − 1);
for i: = 1 step 1 until k − 1 do
    begin swap (p[i], p[k]); permute (k − 1) end;
```

On the assumption that permute $(k-1)$ leaves p unchanged, this kernel has the net effect of rotating the elements $p[1], p[2], \ldots, p[k]$ one place cyclically to the right. This can be seen from the example:

original state:	1	2	3	4	5
after swap ($p[1]$, $p[5]$):	*5*	2	3	4	*1*
after swap ($p[2]$, $p[5]$):	5	*1*	3	4	*2*
after swap ($p[3]$, $p[5]$):	5	1	*2*	4	*3*
after swap ($p[4]$, $p[5]$):	5	1	2	*3*	*4*

Since the overall effect of the operation must be to leave the array p as it was before, the right rotation must be followed by a compensatory left rotation.

```
q: = p[1];
for i: = 1 step 1 until k − 1 do p[i]: = p[i + 1];
p[k]: = q
```

Finally it is necessary to determine an appropriate action for the case where $k = 1$. Recall that the purpose of the procedure is to

"generate all permutations of k objects, issuing a detach command after each of them".

Since the only permutation of one number is that number itself, all that is necessary is to issue a single detach instruction.

The permute procedure must be written as an attribute of the permuter class, so that the detach which it issues relates to the relevant object. The whole class may now be declared:

```
class permuter (n); integer n;
begin integer array p[1:n]; integer q; Boolean more;
    procedure permute (k); integer k;
    if k = 1 then detach else
    begin integer i; permute (k − 1);
        for i: = 1 step 1 until k − 1 do
        begin q: = p[i]; p[i]: = p[k];
            p[k]: = q; permute (k − 1) end;
```

```
            q: = p[1];
            for i: = 1 step 1 until k − 1 do p[i]: = p[i + 1];
            p[k]: = q
        end of permute;
        for q: = 1 step 1 until n do p[q]: = q;
        more: = true; permute (n); more: = false
    end of permuter;
```

Note. The detach issued by a permute procedure instance is *not* an exit out of the procedure instance, and does not return control to the call of the procedure. Rather, it is an intermediate exit out of the object as a whole (including the entire recursion process) and passes control back to the main program which generated or called the object. A subsequent call on the object will thus resume the recursion process exactly where it left off.

The decision (assumption) that the procedure permute should leave the sequence unchanged is really quite arbitrary. The reader is invited to convince himself of this fact by writing a procedure based on the same swapping strategy, which returns with the numbers in the reverse order.

5. List Structures

The facilities introduced above for declaration of classes and reference to objects may be used to represent recursive data structures such as stacks and trees, and even cyclic structures such as two-way lists. This is accomplished by declaring attributes of a class to be references to objects of the very same class.

5.1. Binary Search Trees

A binary tree may be defined as

either (i) **none**
or (ii) a node,

where a node consists of

(a) a left component which is a tree
(b) a right component which is a tree
(c) a val which is an integer.

The val component may be regarded as being associated with each node of the tree. A node whose left and right subtrees are both **none** is a terminal element of the tree (leaf).

A binary search tree is defined as a binary tree which is either **none**, or else it is a node which has a val lying between all vals of its left subtree and all vals of its right subtree, which are themselves both binary search trees. The purpose of a binary search tree is to provide for any integer a swift access to the node which has val equal to that integer; and also to provide swift means of inserting a new node with any given val. Thus a class representing the concept of a binary search tree will have the form:

```
class tree (val); integer val;
  begin ref (tree) left, right;
        procedure insert (x); integer x;
                        .......;
  ref (tree) procedure find (x); integer x;
                        .......;
  end of tree;
```

The bodies of the two procedure components are quite simple recursive procedures, matching the recursive structure of the tree:

```
insert: if x < val then
          begin if left = = none then left: - new tree (x)
                                 else left.insert (x)
          end
  else if right = = none then right: - new tree (x)
                         else right.insert (x);

find: if x = val then this tree
  else if x < val then
          (if left = = none then none
                         else left.find (x))
  else if right = = none then none
                         else right.find (x);
```

In the body of "find" there occurs the expression

```
this tree
```

which is intended to yield as value a reference to the current node, that is, the one which owns this particular instance of the find attribute. For example, if the find procedure of X is called by the function designator

$$X.\text{find } (x)$$

and $X.\text{val} = x$, then the result of the function is the reference value of X itself.

Another operation which is meaningful for a binary search tree is that of scanning all its values in ascending order. This operation may be implemented by a "producing" semicoroutine, which on each call assigns to its attribute

integer current;

the next higher value of a node on the tree. On exhaustion of the tree, the attribute current will take the maximum integer value.

The scanning can be accomplished by a recursive procedure attribute, local to the relevant instance of the coroutine.

```
class scanner (X); ref (tree) X;
    begin integer current;
        procedure traverse (X); ref (tree) X;
            if X =/= none then
                begin traverse (X.left);
                    current: = X.val;
                    detach;
                    traverse (X.right)
                end traverse;
        traverse (X);
        current: = integer max
    end scanner;
```

As an example of the use of these concepts, we consider the task of merging values from several binary search trees, held in an array:

ref (tree) **array** forest [1:*N*];

and outputting the values in ascending order. In order to do this we will require *N* scanners, one operating on each tree of the forest:

ref (scanner) **array** trav [1:*N*];

for *i*: = 1 **step** 1 **until** *N* **do** trav [*i*]: — **new** scanner (forest [*i*]);

Each scanner has now detached with its own minimal val assigned to its own current. All that is now necessary is to locate the minimum of the *N* currents and output it. The corresponding scanner should then be reinvoked to produce its next higher val. When the minimum takes the maximum integer value, the merge is complete.

```
integer min, j, i;
min:= 0;
while min < integer max do
    begin min:= trav [1].current; j:= 1;
        for i:= 2 step 1 until N do
            if min > trav [i].current then
                    begin min:= trav [i].current;
                        j:= i
                    end search for smallest current;
        if min < integer max then
            begin output (min);
                call (trav [j])
            end
    end of merge process;
```

5.2. SYNTAX ANALYSER

As a more substantial example of list processing, we take a general table-driven context-free syntax analyser. We shall use a top-down back-tracking algorithm, which will detect all possible analyses (more than one if the grammar is ambiguous), on condition that the grammar does not contain left recursion. The symbol string is represented by a "tape" with the following operators.

```
procedure move right; ....;
procedure move left; .....;
integer procedure symbol; .....;
```

The "move" operations move a reading head one symbol to the right or left. The "symbol" procedure reads the symbol under the reading head, and converts it to an integer according to a one–one mapping.

A given function "meta" determines whether a given integer represents a meta-symbol.

```
Boolean procedure meta (S); integer S; ...;
```

For simplicity the grammar is represented by a three-dimensional array G

```
integer array G[..., ..., ...];
```

where $G[i, j, k]$ contains the kth symbol of the jth alternative right hand side for the meta-symbol represented by i. There is an

```
integer array jm[...];
```

where *jm*[*i*] is the number of right hand sides for a given meta-symbol. Each right hand side is followed by a special symbol "⊥" outside the vocabulary of the grammar. If one of alternative definitions of the syntactic class is ⟨empty⟩, it will be represented by this symbol alone.

For example, consider a simple context-free grammar for a subclass of arithmetic expressions:

(1) ⟨exp⟩::= ⟨term⟩|⟨term⟩⟨addop⟩⟨exp⟩

(2) ⟨term⟩::= ⟨primary⟩|⟨primary⟩⟨mulop⟩⟨term⟩

(3) ⟨primary⟩::= ⟨constant⟩|⟨variable⟩|(⟨exp⟩)

(4) ⟨addop⟩::= +|−

(5) ⟨mulop⟩::= *X*|/

(6) ⟨constant⟩::= 1|2|3|4|5|6|7|8|9|0

(7) ⟨variable⟩::= *I*|*J*|*K*|*L*|*M*|*N*

There are seven meta-symbols which may be given integer values 1 to 7. The 22 terminal symbols may be given values 8 to 29 inclusive, and the "⊥" terminating symbol may be given value 0.

The array *G* representing this grammar may now be declared:

integer array *G*[1:7, 1:10, 1:4]

The first plane of this array will take the value

G[1, ., .] =	2, 0, 0, 0	first alternative
	2, 4, 1, 0	second alternative
	0, 0, 0, 0	the other 8 rows
	.....	are irrelevant.
	0, 0, 0, 0	

Note also that:

jm[1] = 2, *jm*[6] = 10

meta (1) = meta (7) = **true**

meta (8) = meta (29) = **false**

The desired result obtained by generating an instance of the syntax analyser, with the first symbol of text under the reading head, will be a complete syntax tree representing the text; the character *after* the last character of the analysed text will be under the reading head, and a variable "good" will be set to **true**. Subsequent calls of the same instance will produce trees representing alternative analyses. When no further analyses are possible, the input text will be stepped back to the beginning, and the variable good will be set **false**. This will happen on first generation, if the input text contains a syntax error.

Note that the analyser will discover all successful analyses of any initial segment of the text.

The syntax tree output on each call of the analyser will contain a node for each phrase identified in the text. Each phrase has the following attributes:

integer *i*: indicating the syntactic class of the phrase

integer *j*: indicating which alternative of its class it belongs to

ref (phrase) sub: refers to the last subphrase of the given phrase

ref (phrase) left: refers to the phrase immediately preceding this phrase on the same level of analysis. The left of the first subphrase of any phrase is **none**.

Thus the expression

$$M \times N + 7$$

should give rise to a tree of the form shown in Fig. 1.

The syntax analyser will be recursively structured, as a class of phrase objects, each of which reproduces on a single phrase the intended behaviour of the analyser as a whole.

A phrase object accepts a meta-symbol *i* and a left neighbour as parameter, and is responsible for producing all possible syntax trees of the given syntax class which match a portion of text to the right of (and including) the current symbol. The input text will on each occasion be stepped on to the first symbol which does *not* match the stated analysis. When all possible analyses are complete, the tape is stepped back to the position it was before entry to the given phrase, a global variable good is set to **false**, and the phrase terminates.

We are now in a position to outline the general structure of the phrase class:

```
class phrase (i, left); integer i; ref (phrase) left;
    begin integer j; ref (phrase) sub;
        for j := 1 step 1 until jm[i] do
            ...match remainder of text in all possible
               ways to alternative j of class i,
               issuing a detach after each successful match...;
        good := false
    end of phrase;
```

Assume that an object has successfully matched the first $k - 1 (k > 0)$ symbols of a chosen alternative (j) for the given meta-symbol (i). We now formulate a piece of program for matching the kth symbol to the input in all possible ways. We assume that the remainder, if any, of the right hand side is

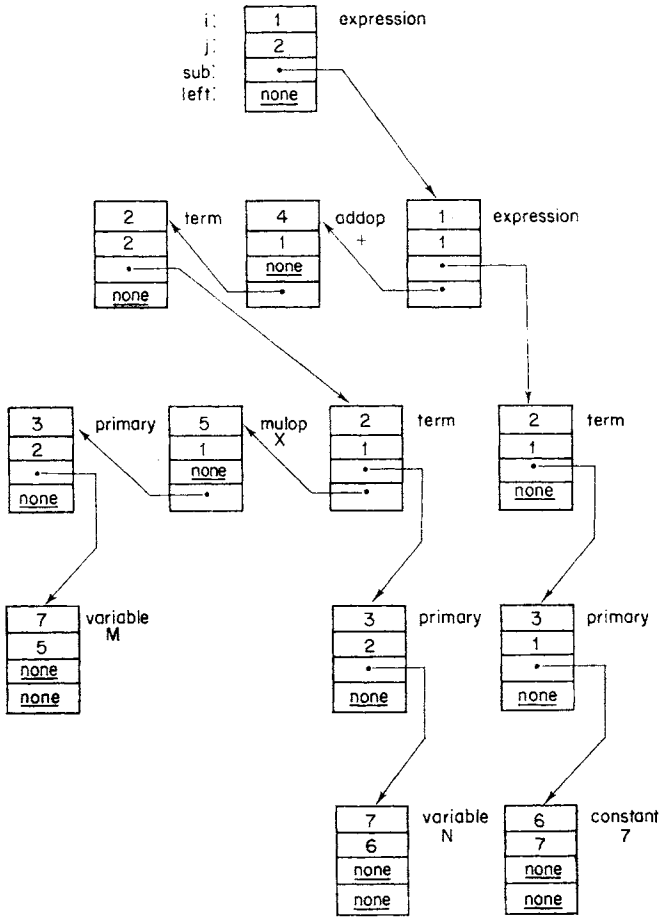


FIGURE 1

matched to the input in all possible ways by the statement "match remainder", and that this statement leaves unaltered the position of the reading head and the part of the syntax tree so far constructed. We make the latter assumption also for an object which has failed to identify (another) phrase.

1. **begin integer** g; $g := G[i, j, k]$;
2. **if** $g =$ "$\perp$" **then begin** good: = **true**; detach **end**
3. **else if** $g =$ symbol **then**
4. **begin** move right; match remainder; move left **end**
5. **else** if meta (g) **then**
6. **begin** sub: – **new** phrase (g, sub);

```
7.          while good do
8.          begin match remainder; call (sub) end;
9.          sub: — sub.left
10.     end
11. end
```

Comments.

Line 1. The kth symbol of the right hand side number j is called g for brevity.

Line 2. If g is the terminator symbol the whole right-hand side has been successfully matched to the input. The object reports back to its master. Line 2 does not alter the syntax tree or the position of the reading head.

Line 4. Since we have a match the object moves the reading head to the next symbol. After having matched the remainder in all possible ways the object restores the position of the reading head. Thus, according to assumptions, line 4 has a null net effect.

Line 6. Since g is a meta-symbol, a new phrase object is generated to identify sub-phrases of the syntax class g. It becomes the new rightmost sub-phrase. Its left neighbour phrase is the old rightmost sub-phrase.

Line 7. We have assumed that an object when failing sets "good" to **false**.

Line 8. Since "good" is **true**, a sub-phrase has been identified matching g. After having matched the remainder in all possible ways, "sub" is called to identify the next possible sub-phrase. Since we want to match g in all possible ways, line 8 is repeated until the sub-phrase object fails.

Line 9. According to assumptions a phrase object which has failed, has had a null net effect. The total effect of lines 6–8 is thus to add an object to the syntax tree. Line 9 restores the syntax tree to its original state.

The comments show that the block above matches the symbol g followed by the remainder of the jth right-hand side of i in all possible ways and has a null net effect. Consequently the block itself satisfies the assumptions made for the "match remainder" statement. It follows that the whole matching algorithm may be expressed in a simple way by a recursive procedure. The whole computing process is described by the following class declaration.

```
class phrase (i, left); integer i; ref (phrase) left;
begin integer j; ref (phrase) sub;
    procedure match (k); integer k;
    begin integer g; g := G[i, j, k];
        if g = "⊥" then begin good := true; detach end
        else if g = symbol then
        begin move right; match (k + 1); move left end
        else if meta (g) then
        begin sub :- new phrase (g, sub);
                    while good do
                    begin match (k + 1); call(sub)end;
                    sub :- sub.left
        end
    end of match;
    for j := 1 step 1 until jm[i] do match (1);
    good := false
end of phrase
```

A master program could have the following structure

```
ref (phrase) tree;
. . . . . .
tree :- new phrase (start, none);
while good do begin found: . . . . . ; call(tree)end;
```

where "start" represents the start symbol of the grammar. At the label "found" a sentence has been identified and the variable "tree" refers to its syntax tree represented as described above. For each node its associated meta-symbol (i), the rhs alternative number (j), and the links to other nodes (sub, left) are available through remote identification, for example

tree.i, tree.sub.j, tree.sub.left.left.sub.

We must expect in general that the strings matched by different successful trials may be of unequal lengths, starting at the same location of the tape. This may be avoided by defining the language in such a way that no initial segment of a valid text is also valid. Alternatively, the whole text should be

followed by some symbol outside the alphabet, say "⊥", and the master program might have the following structure

```
ref(phrase)parse; Boolean good;
parse: – new phrase (start, none);
while good do
begin if symbol = "⊥" then inspect successful parse;
    call(parse)
end
```

It is a remarkable feature of the phrase class that the result it yields on each call is a tree whose nodes consist in phrase objects which have been activated recursively and not yet terminated. Each of these phrase objects plays a dual role, both as a part of the syntactic tree which is to be inspected by the master program, and as the set of local variables for the recursive activations of other phrase objects. It is this close association of data and procedure which permits the algorithm to be so simply and concisely formulated.

Notice that each phrase object is the nucleus of a separate stack of recursive activations of its local "match" procedure. At the time when a detach is issued on behalf of an object, signalling a successful (sub-) parse, its stack has attained a temporary maximum depth, one level for each symbol in the current right-hand side, plus one level corresponding to the rhs terminator ⊥, which issued the detach.

Thus the whole dynamic context of a successful parse is preserved. When an object is called to produce an alternative parse a backtracking process takes place, during which the "match" stack of the object is reduced. At a level corresponding to a meta-symbol in the rhs the match procedure calls on the corresponding phrase object to produce an alternative sub-parse (line 8) and so on. (cf. the row of officers in Chaplin's Great Dictator).

6. Program Concatenation

In the preceding sections we have seen how the *class* mechanism is capable of modelling certain simple concepts, by specifying data structures and defining operations over them. In this section, we develop a method by which more elaborate concepts can be constructed on the basis of simpler ones. This will establish potential hierarchies of concepts, with complex concepts subordinate to the more simple ones in terms of which they are defined. The structuring technique gives a new method of composing a program from its constituent parts, and is known as *concatenation.*

Concatenation is an operation defined between two classes A and B, or a class A and a block C, and results in formation of a new class or block. Concatenation consists in a merging of the attributes of both components, and the composition of their actions. The formal parameters of the concatenated object consist of the formal parameters of the first component followed by formal parameters of the second component; and the same for specification parts, declarations and statements (if any).

A concatenated class B is defined by prefixing the name of a first component A to the declaration of B:

A **class** $B(b_1, b_2, \ldots)$; ... specification of b's;
begin...attributes of B...; ...actions of B...**end**

Suppose the class A has been defined:

class $A(a_1, a_2, \ldots)$; ...specification of a's...;
begin...attributes of A...; ...actions of A...**end.**

According to the concatenation rules, the effect of the prefixed declaration for class B is the same as if B had been declared without a prefix thus:

class $B(a_1, a_2, \ldots, b_1, b_2, \ldots)$; ...specification of a's...
specifications of b's...;
begin...attributes of A...; ...attributes of B...;
...statements of A...; ...statements of B...**end**;

Note. If any local identifiers of A are the same as local identifiers of B, the collision of names is resolved by systematic change of B's identifiers. A block also may be prefixed by a class identifier:

A **begin**...declarations...; ...statements **end**,

and the effect is similar to that described above, except that the result of the concatenation is a block, not a class. (If the class A has parameters, the prefix must include an actual parameter part 1). The effect of prefixing a block is to make available within that block the library of procedures and related data (including even classes) declared within the class declaration for A.

A single class may be used as prefix to several different concatenated classes. For example, suppose a program requires to deal with trucks, buses, and private cars. These are three separate classes, and each has its own attributes. But there are certain attributes (for example license number) which are common to all of them, by virtue of the fact that they are all vehicles. The concept of vehicle is a more general one, and could be declared as a separate concept;

class vehicle (license no); **integer** license no; ...;

This class can now be used as a prefix in the remaining class declarations.

vehicle **class** truck (load); **real** load; ...;

vehicle **class** bus (seating); **integer** seating; ...;

vehicle **class** car; ...;

An object belonging to a prefixed class is a compound object, which has certain attributes and operations *in addition to* those defined in the prefix part. Thus a truck object has a license no and a load, and a bus object has a license no and a seating. It is reasonable to regard "truck" and "bus" and "car" as *subclasses* of the vehicle class; any object of a subclass also belongs to the prefix class "vehicle".

A reference variable may be qualified as belonging to a prefix class or to a concatenated class. If it belongs to the prefix class it may point to objects of *any* of the subclasses, and may be used in a remote identifier to access any of the attributes of the prefix class but not to access any attributes of the subclasses. A reference qualified by a subclass may point only to objects of the subclass, but may be used in a remote identifier to access all its attributes. Thus given the reference variables:

ref (vehicle) V; **ref** (bus) B;

the following are valid remote identifiers:

V.license no, B.license no, B.seating,

but V.seating is not valid.

Thus the subclass notion provides a useful flexibility of object referencing. A "weak" qualification permits a wide range of objects referencing, at the cost of inability to make remote access to attributes declared in a subclass.

Assignment of a subclass reference to a prefix class reference variable (e.g. $V:-B$) is always valid, and can be recognised as such at compile time. But assignment in the other direction (e.g. $B:-V$) may give rise to an error (detected only at run time), if the object referenced does not in fact belong to the expected subclass (bus).

6.1. BINARY SEARCH TREE

Suppose it is desired to set up a binary search tree to hold information about stock items in an inventory. Each node of the tree should contain not only a val (indicating the stock number of the item) but also certain other information about quantity on hand, price, reorder point, etc. The simplest way of achieving the required effect is to prefix the class "stock item" by the class tree, and then declare the additional attributes required, for example:

```
tree class stock item;
    begin integer qoh, price, reorder point;
        Boolean ordered;
        procedure reduce;
            begin if qoh = reorder point & ¬ordered then
                                issue reorder;
                        qoh: = qoh − 1
            end of remove;
    end of stock item;
```

6.2. TWO WAY LIST

Taking advantage of the concatenation technique, it is possible to design classes which are intended solely or primarily to act as prefixes to other classes or to blocks. In this section we give an example of a class TWLIST, which is intended to be used as a prefix to a block, and to make available within that block the concept of two-way chained cyclic lists. Such a list consists of a *list head*, which contains two pointers, one to the first element of the chain and one to the last. Each *link* in the chain must also contain two pointers, suc which points to the successor in the list (or the list head if there is none), and pred which points to the predecessor in the list (or the list head if there is none). In an empty list, the two pointers from the list head point to the list head itself.

Each pointer in the system must be capable of pointing either to another link in the list or to a list head. Therefore these pointers must be qualified by a class which embraces both links and heads, i.e. a class "linkage" of which they are both subclasses. Since both list heads and links require two reference attributes, suc and pred can be declared as attributes of the prefix class linkage.

The single concept of a two-way list is represented by the triple of classes linkage, link, and list head. In order to indicate that they are to be considered in conjunction as a single concept, the declarations for all three classes are grouped together in a single class declaration TWLIST, which is to be used as a prefix to any block which requires to use the concept. Within such a block the "link" class is intended to be used as a prefix to other classes specifying the nature of the items; for example, if stock items were to be held in a two-way list instead of in a binary search tree, the declaration would be:

```
link class stock; ...as before...;
```

It is now necessary to decide on a basic set of operations on lists and links. A link I should be removable from its list by a procedure statement

I.out;

and it should be capable of being reinserted in a list just before link J by a procedure statement:

I.precede (J);

Since a link can belong to at most one list, this operation should also remove I from any list it happens to belong to before. Finally a link should be insertable as the last item of a list with head H by a procedure statement:

I.into (H);

For a list head H, it seems useful to define the following functions

H.empty,

which tests whether the list is empty,

H.first

which yields H's first item, if any; otherwise **none**, and

H.last

which yields H's last item, if any; otherwise **none**.

The declaration of the class TWLIST can now be given:

```
1.  class TWLIST;
2.  begin class linkage; begin ref (linkage) suc, pred; end;
3.      linkage class link;
4.      begin procedure out;
5.          if suc =/= none then
6.          begin suc.pred:- pred; pred.suc:- suc;
7.              suc:- pred:- none
8.          end of out;
9.          procedure precede (x); ref (linkage) x;
10.         begin out; suc:- x; pred:- x.pred;
11.             suc.pred:- pred.suc:- this link
12.         end of precede;
13.         procedure into (L); ref (list) L;
14.             precede (L);
            comment suc and pred of a link object should have the
            standard initial value none indicating no list membership;
15.     end of link;
```

```
16.     linkage class list;
17.     begin ref (link) procedure first;
18.             first: - if empty then none else suc;
19.         ref (link) procedure last;
20.             last: - if empty then none else pred;
21.         Boolean procedure empty;
22.             empty: = suc = = this list;
23.         suc: - pred: - this list
                comment suc and pred of a list head should be
                initialized to indicate an empty list;
24.     end of list;
25. end of TWLIST;
```

Let P be an *arbitrary* block instance prefixed by TWLIST, which, outside its prefix part, contains no explicit reference assignment to any variable suc or pred of any linkage object. Then the assertions (1) and (2) below are valid throughout the lifetime of P (at times when control is textually outside the body of TWLIST).

(1) Any linkage object x in P is either an object with no list membership, in which case $x.\text{suc} == x.\text{pred} ==$ **none** and $x \notin$ list, or $x.\text{suc}.\text{pred} == x.\text{pred}.\text{suc} == x$.

It follows that all lists contained in P are circular. Furthermore:

(2) Each circular list in P contains exactly one list head, which is an object of the class "list".

The assertions are established by observing that each of the operations below preserves their validity, and that P contains no linkage object initially.

new *link* (or **new** C, $C \subseteq$ linkage–list) generates a link object, which is not a list member (its suc and pred are automatically initialised to **none**).

new *list* (or **new** C, $C \subseteq$ list) generates an "empty" circular list containing the generated list head and initially nothing else.

In the following we assume $x \in$ link, $y, z \in$ linkage, and $L \in$ list. $x \leftrightarrow y$ is an abbreviation for $x.\text{suc} == y \;\&\; x == y.\text{pred}$.

$x.out$ If $z \leftrightarrow x \leftrightarrow y$ the result is $z \leftrightarrow y$ and x is not a list member. (Notice that (2) together with $x \in$ link implies $x =/= y, z$.) If x was not a list member, the result is to do nothing.

$x.precede$ (**y**), where $x =/= y \wedge z \leftrightarrow y$. The result is $x \leftrightarrow y$ (and $z \leftrightarrow x$ if $x =/= z$). If x was a list member, x is first removed from that list.

$x.into$(L), where $z \leftrightarrow L$. The result is $z \leftrightarrow x \leftrightarrow L$, which implies $x == L.$last. If x was a list member, x is first removed from that list.

Any use of out, precede, or into not satisfying the above assumptions, is either textually illegal or leads immediately to a run time error and program termination caused by an invalid remote identifier. E.g. the operation $x.$precede (y) sets $x.$pred to **none** if $x == y$ or y is not a list member. Consequently the remote identifier pred.suc in the body of precede is invalid. Notice that $x.$into (L) is a "safer" operation, since $x \in$ link, $L \in$ list implies that $x =/= L$ and $L.$pred $=/=$ **none**.

The assertions (1) and (2) provide a guarantee that our lists are well behaved, provided that no explicit assignment to any variable suc or pred occurs. The construction TWLIST is thus a reliable "mental platform," which in a certain sense *cannot break down*, whatever programming errors are made. When programming on top of TWLIST one is entitled to ignore the list processing details involved in manipulating the circular two-way lists. Each list object may be regarded as representing an *ordered set* of link objects, with the proviso that a link object may be member of at most one such set at a time. The last fact is reflected in the design of the procedures into and precede. Explicit use of the attributes suc and pred, e.g. for scanning through a list, may, however, require the user to be conscious of the fact that the "last" member has a successor and the "first" member a predecessor, which are both identical to the list object itself. A design alternative is to suppress this fact by declaring the following procedures as attributes to link.

```
ref (link) procedure successor;
    inspect suc when list do successor: - none
    otherwise successor: - suc;
ref (link) procedure predecessor;
    inspect pred when list do predecessor: - none
    otherwise predecessor: - pred;
```

Note the construction

inspect *r* **when** *C* **do**...

enables the programmer to test whether the object referenced by *r* belongs to one of its possible subclasses *C*.

7. Concept Hierarchies

At the outset of a programming project there is a *problem*, more or less precisely defined and understood in terms of certain problem oriented concepts, and a *programming language*, perhaps a general purpose one, providing some (machine oriented) basic concepts, hopefully precisely defined and com-

pletely understood. There is a *conceptual distance* between the two, which must be bridged by our piece of program. We may picture that distance as a vertical one, the given programming language being the ground level.

Our difficulty in bridging such gaps is the fact that we have to work sequentially on one simple part problem at a time, not always knowing in advance whether they are the *right problems*.

In order to better overcome such difficulties we may build pyramids. Unlike the Egyptian ones ours are either standing on their heads (bottom-up construction) or hanging in the air (top-down construction). The construction principle involved is best called *abstraction*; we concentrate on features common to many phenomena, and we abstract *away* features too far removed from the conceptual level at which we are working. Thereby we have a better chance of formulating concepts which are indeed useful at a later stage.

In the bottom-up case we start at the basic language level and construct abstract concepts capable of capturing a variety of phenomena in some problem area. In the top-down case [8, 9] we formulate the solution to a given problem in terms of concepts, which are capable of being implemented (and interpreted) in many ways, and which are perhaps not yet fully understood. In either case system construction may consist of adding new layers of pyramids (above or below) until the conceptual gap has finally been bridged. Each such layer will correspond to a conceptual level of understanding.

For instance, given some problem which involves queueing phenomena, we could take TWLIST of the preceding section as the first step of a bottom-up construction. Then, for the remainder of the construction we are free to think and express ourselves in terms of dynamic manipulation of ordered sets of objects.

Layers of conceptual levels may be represented as a *prefix sequence* of class declarations. For example, it is possible to construct a series of class declarations, each one using the previous class as prefix

$$\textbf{class } C_1; \ldots.;$$
$$C_1 \textbf{ class } C_2; \ldots.;$$
$$\ldots\ldots$$
$$C_{n-1} \textbf{ class } C_n; \ldots.;$$

The list $C_1, C_2, \ldots, C_{n-1}$ is known as the prefix sequence for C_n. The outermost prefix C_1 is built at the ground level. Every other level rests on the one(s) below, in that it may take advantage of all attributes of its entire prefix sequence. Making use of this language mechanism, bottom-up construction of a program is to plan and write the classes of a prefix sequence one by one in the stated order. The program itself is finally written as a prefixed block on top of the whole sequence.

$$C_n \textbf{ begin} \text{ ——— } \textbf{end}$$

The top–down strategy would correspond to constructing the members of the prefix sequence, including the prefixed block, in the reverse order. (SIMULA 67 contains additional mechanisms, not considered here, for facilitating top–down and mixed mode construction.)

A well-formed conceptual level (bottom-up) is a set of well-defined inter-related concepts, which may be combined to make more elaborate concepts. It may serve for further construction as a mental platform, raised above ground towards some application area, i.e. as an "application language". A preconstructed application language may serve to shorten the conceptual gap that has to be bridged for many problems in the area. The usefulness of such a platform is closely related to its ruggedness, that is with the way in which it tolerates or even forestalls misuse. As we saw in the last section TWLIST supplies an exceptionally rugged mental platform; and in this section we shall build on it a small but useful application language, which may in its turn be used as a platform for the solution of realistic problems.

7.1. DISCRETE EVENT SIMULATION

Simulation is a method for studying the behaviour of large systems of interacting objects, and evaluating the effect of making changes which would be too expensive to make on an experimental basis in real life. The object of a simulation model could be a production line, a traffic system, a computer system (hardware and software), a social system composed of interacting individuals, etc. The following notions are common to most such systems.

(1) Processes taking place in *parallel*, giving rise to discrete events at irregular intervals of time.

(2) *Queueing* phenomena, arising when an object has to wait for service from a currently busy server.

In order to represent processes occurring in parallel, it is not necessary that the corresponding program components should be multiprogrammed in the computer; but it is necessary that the programs should be able to suspend themselves temporarily, and be resumed later from where they left off. Thus the active objects or "processes" in a simulation will be represented by (semi-)coroutines, operating in *pseudo–parallel* under control of a scheduling mechanism.

For example, in a job shop simulation, an incoming order gives rise to a sequence of events on the shop floor, to satisfy the order. Each order may be regarded as a process whose activity is to proceed from one machine to the next, requesting and obtaining service from it. The sequence of requests is determined by the nature of the order. If the requested machine is free, the order is served immediately, and the machine goes busy for a period equal to the length of the service. Otherwise, the order joins a queue of orders waiting for the machine to become free.

In the implementation of the concept of simulated time, the first requirement is that each process have access to a variable "time" which holds the current time, and which is incremented on appropriate occasions by the time-control mechanism. Note that the updating of this variable must be entirely independent of the passage of computing time during the simulation, since actions which take a long time on a computer might take only a short time in the real world, and vice versa. As far as simulated time is concerned, the active phases of the processes must be instantaneous; "time" does not move until all the participating processes are passive.

Thus in order to simulate the passing of time, a process simulating an active system component must relinquish control for a stated interval T of simulated time; and it must be reactivated again when the time variable has been incremented by T. This will be accomplished by the process calling the procedure

hold (T).

For example, an order which has found its required machine ready to serve it needs to indicate how long this service will take, by the statement

hold (service interval);

The order will now become inactive until all other orders which were due to be reactivated *before* time + service interval have been reactivated, and have relinquished control again. At this point, the given order will be reactivated, and will find that its time has been appropriately incremented.

While a process is held, it will be necessary to record its reactivation time as one of its attributes. It is convenient therefore to use the time attribute of the process itself for this purpose.

The method of holding for a specified interval is possible only if the process knows how long it has to wait before the next "event" in its life. But sometimes it may require to wait until the occurrence of some event in the life of some other process. For example, an order, on finding its required machine busy, must join a queue and wait until the machine is free; and an order on releasing a machine must activate the first other order in the queue (if not empty). Thus two additional procedures are required:

wait (Q),

and activate (X),

where Q refers to the queue (two-way list) on which the calling process is to wait while it is passive, and X refers to some passive process, which is to be removed from its queue and allowed to proceed.

Finally, a means must be provided of starting and stopping the simulation. This may be accomplished by a procedure statement

simulate (start, finish),

where start refers to the process with which the simulation starts, and finish gives the time limit for the simulation. Any process requesting to be held beyond this limit may be ignored. Presumably, the start process will activate other processes to participate in the simulation.

We now proceed to implement the mechanism described above. It will be implemented as a class MINISIM, which is intended to be used as a prefix to a simulation main program. In order to take advantage of the two-way list mechanism, the MINISIM class must be prefixed by TWLIST. This ensures that TWLIST is also available in any simulation program which is prefixed by MINISIM.

A class of objects which are to be capable of participating in a simulation should be declared as a subclass of the "process" class. This will make available to it the necessary time control and queueing mechanisms. Each process must have the capability of inserting itself into a two-way list; therefore the process class itself must be declared as a subclass of the class of links.

Processes waiting for the elapse of their holding interval are held on a unique two-way list known as the sequencing set (*SQS*). The processes are ordered in accordance with decreasing reactivation times. A specially created finish process is always the first link in *SQS*, and the last link is always the one that is currently active. Its time represents the current time of the system. When it goes inactive, its predecessor in the *SQS* will (usually) become the last, and its local time has already been updated to the time at which that process was due to be reactivated.

We are now in a position to give the general structure of MINISIM, omitting for the time being the procedure bodies.

```
TWLIST class MINISIM
  begin ref (list) SQS;
      ref (process) procedure current;
              current:- SQS.last;
    link class process;
      begin real time;
          procedure hold (T); real T;
                  .....;
          procedure wait (Q); ref (list) Q;
                  .....;
          procedure activate (X); ref (process) X;
                  .....;
      detach; comment a new process doesn't actually
```

```
                    do anything until it is activated
                end of process;
            procedure simulate (start, finish);
                    ref (process) start; real finish;
                    .....;
        end of MINISIM.
```

We shall give the bodies of the procedures in reverse order.

```
    simulate: begin SQS: - new list;
                new process.into (SQS); current.time: = finish;
                if start.time < finish then start.into (SQS);
                while ¬SQS.empty do
                    begin call (current);
                        current.out;
                        comment this ensures that a terminated
                        or detached process leaves the SQS;
                    end
            end of simulate
    wait: begin into (Q); resume (current) end;
```

The active process inserts itself into the queue, and thereby leaves the *SQS*. It also resumes the process on the *SQS* which is next due to be reactivated. Notice that the standard sequencing mechanism of the simulate procedure must be bypassed, since the old active process already is out of the *SQS*.

```
    activate: begin X.into (SQS); comment as its last and current member;
                X.time: = time; comment i.e. now;
                resume (current);
            end of activate.
```

The calling process places *X* ahead of itself in *SQS*, but with the same time. Since the calling and the activated process *X* have the same time, it does not matter to the timing mechanism in what order they are placed; our choice implies that an active phase of *X* is invoked immediately in real time. Control returns to the calling one at the same moment of simulated time, but after the completion of the active phase of *X*.

```
hold:
begin ref (process) P;
        P:- pred;
        comment the holding process is necessarily active, and
          therefore also the last member of SQS. Since the
          finish process never holds, there will always be a
          second-to-last process on SQS
        if T > 0 then time:= time + T;
        comment set local reactivation time,
          time should never decrease;
        if time ⩾ P.time then
          begin comment this process must be moved in SQS;
                out; comment of SQS, now P is current;
                P:- SQS.first; comment the finish process;
                if time < P.time then
                    begin comment reactivation time is
                          within the time limit;
                          while time < P.time do P:- P.suc;
                          comment terminates since
                                  time ⩾ current.time;
                          precede (P)
                    end; comment ignore a process that would
                          exceed the limit;
                  resume (current)
          end;
    end of hold;
```

Notice that a process object is allowed to relinquish control simply by saying detach or by passage through its **end**. In both cases control returns to the standard sequencing mechanism of the simulate procedure. The basic activation instructions call and resume, however, should not be explicitly applied to process objects; that would illegally bypass the timing mechanism.

7.2. THE LEE ALGORITHM

As a simple but unexpected example of the use of simulated time, we take the Lee algorithm for finding the shortest path between a city A and a city B connected by a network of one-way roads. The algorithm may be envisaged

as the propagation of a pulse from the destination city *B* at equal speed along all roads leading into it. Each time a pulse reaches a city not reached by a previous pulse, it records the road it has come along, and then sends pulses outward along all roads leading into the city. When a pulse reaches a city which has already been reached by another pulse, it dies. When a pulse reaches the city *A*, the task is completed.

Cities and roads may be represented by classes.

```
class city; begin ref (road) roadsin, wayout; ....end
class road; begin real length; ref (road) nextin;
                    ref (city) source, destination; ...end
```

The variable wayout holds the recommended wayout from the city towards *B*. For an unvisited city, its value is **none**.

The class representing a pulse takes as parameter the road along which it is first to pass.

```
process class pulse (rd); ref (road) rd;
begin ref (city) c; c:- rd.source;
    hold (rd.length);
    if c.wayout =/= none then
    begin c.wayout:- rd;
        if c == A then go to done;
        comment stops the simulation by
                going to a non-local label;
        rd:- c.roadsin;
        while rd =/= none do
        begin activate (new pulse (rd));
            rd:- rd.nextin
        end propagation of pulses
    end
end of pulse
```

The algorithm will be invoked by calling a procedure with parameters indicating the starting and final cities, and an upper limit *L* on the length of the path that is to be printed. It is assumed that the wayout of every city is initially **none**. The time and process concepts are made available by the prefix MINISIM to the procedure body.

```
procedure Lee (A, B, L); ref (city) A, B; real L;
    MINISIM begin process class pulse (rd); ref (road) rd;
                        ...as before...;
                    process class starter;
                        begin ref (road) rd;
                            rd:- B.roadsin;
                            while rd =/= none do
                                begin activate (new pulse (rd));
                                    rd:= rd.nextin
                                end
                        end of starter;
                        simulate (new starter, L);
        done:end of Lee;
```

After a procedure statement such as

```
Lee (Oslo, Belfast, 1000);
```

where **ref** (city) Oslo, Belfast; the required route may be printed out, provided that it exists.

```
if Oslo.wayout =/= none then
begin ref (city) c; procedure print.....; .....;
    print (Oslo.wayout); c:- Oslo.wayout.destination;
    while c =/= Belfast do
    begin print (c); print (c.wayout);
        c:- c.wayout.destination
    end
end else outtext ('no road connection within limit);
```

It is assumed for the print procedure that cities and roads are objects belonging to a common class, by having the same prefix to the two classes. The prefix part of an object might contain the necessary identifying text, such as 'London' or 'M1' as data.

7.3. A JOB SHOP MODEL

As a second application of MINISIM we shall design a model of a simple job shop system. The model may be used to evaluate the capacity of the shop in relation to a given order load. The line numbers below refer to the program on page 218.

The system consists of *machine groups* (lines 3–10), numbered from 1 to *nmg* (lines 1, 11), and *order* objects (lines 12–22). The machines of a group are identical and therefore need not be represented individually; however, their number is specified initially by the value of the attribute *nm* (line 3). Associated with the group is also a queue of orders waiting to be processed, which is empty initially (lines 4, 9), and procedures to *request* a machine for processing (lines 5–6) and to *release* it when finished (lines 7–8).

The variable *nm* is used to represent the number of available machines, say m, as well as the number of orders, say w, waiting in the queue, as described by the following assertion.

$$\textbf{if } nm > 0 \textbf{ then } m = nm \wedge w = 0$$
$$\textbf{else } m = 0 \wedge w = abs(nm)$$

The assertion is valid for each machine group (outside the procedure bodies request and release).

When a machine is requested and $m = 0$, the caller must enter the queue and wait for its turn (line 6). When a machine is released and $w \neq 0$, one of the waiting processes should proceed. The first member of the queue is activated and thereby leaves the queue. The queueing discipline is thus first come first served.

The orders are process objects, each of which generates its successor, (line 18) and which goes from one machine group to the next (lines 20–21) according to an individually defined schedule. For a given order the schedule has n steps, and for each step $s (s = 1, 2, \ldots, n)$ a machine group number ($mg[s]$) and an associated processing time ($pt[s]$) are given. Thus the order should spend the time $pt[s]$ in being processed at machine group number $mg[s]$ (line 21, hold). Notice that the request statement of line 21 will require some additional amount of simulated time for its completion, if the group *m*group [$mg[s]$] currently has no available machine.

The model is driven by input data. In particular, each order object during its first active phase reads in its own schedule, consisting of length of schedule (line 18) arrival time (inreal, line 15), and the values of *mg* and *pt* (lines 16–17). The main program sets up machine groups of specified sizes (lines 24, 25) and generates the first order at time zero. (The procedures inint, inreal, and lastitem are procedures associated with a standard input file, which is part of the program environment).

It is assumed that the input file starts with the following data:

$$nmg, \text{ timelimit}, nm_1, nm_2, \ldots, nm_{nmg},$$

defining the structure of the job shop; and this is followed by an occurrence of

$$n, T, mg_1, pt_1, mg_2, pt_2, \ldots, mg_n, pt_n,$$

for each order to be generated. Each value T defines the arrival time of the order. It is assumed that the T values are in a non-decreasing sequence.

The JOB SHOP goes as follows.

```
1.  begin integer nmg; nmg:= inint;
2.  MINISIM begin
3.  class machine group (nm); integer nm;
4.  begin ref (list) Q;
5.      procedure request;
6.      begin nm:= nm - 1; if nm < 0 then current.wait (Q) end;
7.      procedure release;
8.      begin nm:= nm + 1; if nm ≤ 0 then current.activate (Q.first) end;
9.      Q:- new list
10. end of machine group;
11. ref (machine group) array mgroup [1:nmg];
12. process class order (n); integer n;
13. begin integer array mg[1:n]; array pt[1:n]; integer s;
14.     ref (machine group) M;
15.     hold (inreal-time); comment arrival time is now;
16.     for s:= 1 step 1 until n do
17.     begin mg[s]:= inint; pt[s]:= inreal end;
18.     if ¬lastitem then activate (new order (inint));
19.     comment generate next order, if any;
20.     for s:= 1 step 1 until n do
21.     begin M:- mgroup [mg[s]]; M.request; hold (pt[s]);
                                                M.release end
22. end of order;
23. integer k; real lim; lim:= inreal;
24. for k:= 1 step 1 until nmg do mgroup [k]:- new machine group (inint);
25. simulate (new order (inint), lim);
26. comment initial time is zero by default;
27. end of program;
```

The model above should be augmented by mechanisms for observing its performance. We may for instance very easily include a "reporter" process, which will operate in "parallel" with the model components and give output of relevant state information at regular simulated time intervals.

```
process class reporter (dt); real dt;
while true do
begin hold (dt);
      give output, e.g. of
      mgroup [k].nm ∨ (k = 1, 2, ..., nmg)
end of reporter;
```

The first order could generate a reporter object and set it going at system time zero.

```
activate (new reporter (inreal))
```

Output will then be given at system time t, $2t$, $3t$, ..., where t is the actual parameter value.

As a further example we may wish to accumulate for each machine group a histogram of waiting times of orders at the group. Then define the following subclass of machine group, redefining the operation "request".

```
machine group class Machine Group;
begin ref (histogram) H;
      procedure request;
      begin real T;
            T:= time; nm:= nm − 1;
            if nm < 0 then wait (Q);
            H.tabulate (time − T)
      end of new request;
      H:− new histogram (X, N)
end of Machine Group;
```

It is assumed that "histogram" is the class defined in section 3.1, and that **array** $X[1:N]$ and **integer** N are nonlocal quantities. Now replace the lower case initials by upper case in the class identifier of lines 11, 14, and 24. Then all machine groups will be objects extended as above, and since the qualification of the reference variable M is strengthened, the "request" of line 21 refers to the new procedure. Thus a histogram of waiting times will be accumulated for each group.

Finally it should be mentioned that the "machine group" concept might have considerable utility as a general purpose synchronisation mechanism for

pseudo–parallel processes. It might be useful to phrase it in more abstract terminology and possibly include it as part of a "third floor" platform for "resource oriented" simulation. In fact well known special purpose languages [10, 11] have elaborations of this concept ("facility", "store") as fundamental mechanisms. The analogy to the semaphore mechanism [12] for the synchronisation of truly parallel processes should be noted. The procedures request and release correspond to the *P* and *V* operations, respectively.

REFERENCES

(1) Naur, P. (ed.) (1962/63). Revised Report on the Algorithmic Language. ALGOL 60. *Comp. J.*, **5**, pp. 349–367.

(2) Dahl, O.-J., Myhrhaug, B., Nygaard, K. (1968). The Simular 67 Common Base Language. Norwegian Computing Centre, Forskningsveien 1B, Oslo 3.

(3) Wang, A., Dahl, O.-J. (1971). Coroutine Sequencing in a Block Structured Environment. *BIT* **11**, 4, pp. 425–449.

(4) Dahl, O.-J., Nygaard (1966). Simula—an Algol-Based Simulation Language. *Comm. A.C.M.* **9**, 9, pp. 671–678.

(5) Dahl, O.-J. (1968). Discrete Event Simulation Languages. "Programming Languages" (ed. Genuys, F.). pp. 349–395. Academic Press, London.

(6) Hoare, C. A. R. (1968). Record Handling. "Programming Languages" (ed. Genuys, F.). pp. 291–347. Academic Press, London.

(7) Conway, M. E. (1963). Design of a Separable Transition—Diagram Compiler. *Comm. A.C.M.* **6**, 7, pp. 396–408.

(8) Naur, P. (1969). Programming by Actions Clusters. *BIT* **9**, 3, pp. 250–258.

(9) Dijkstra, E. W. (1972). Notes on Structured Programming. "Structured Programming". pp. 1–82. Academic Press, London.

(10) Knuth, D. E., McNeley, J. L. (1964). SOL—A Symbolic Language for General-Purpose Systems Simulation. IEEE Trans. E.C.

(11) IBM, General Purpose Systems Simulator.

(12) Dijkstra, E. W. (1968). Co-operating Sequential Processes. "Programming Languages". pp. 43–112. Academic Press, London.