1-1-1984

A study of the simula 67 language.

Andrew Joseph Tanhauser

A STUDY OF THE SIMULA 67 LANGUAGE

by

Andrew Joseph Tanhauser

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computing Science

Lehigh University

1984

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

May 3, 1984
Date

Professor in Charge

Head of Division

TABLE OF CONTENTS

ABSTRACT

SIMULA 67, a direct offshoot of SIMULA I, is a general purpose language whose main application area has been simulation. The objectives of SIMULA I was for system description and simulation. The process concept was the basic feature. Various shortcomings of SIMULA I (Nygaard and Dahl, 1982) led to the creation of SIMULA 67 which had the same basic objectives of SIMULA I. However, SIMULA 67 was also to be a general purpose programming language. SIMULA 67 allows for the definition of abstract data types. Class instances may coexist at run-time. References to the classes enable access to their individual components. Overall, SIMULA 67 preserves the ALGOL-like tree structure of programs and allows for ALGOL-like control structures and procedures. Concurrent execution is simulated by using coroutines. Procedural abstractions are supported and data abstraction can be implemented. Hierarchical program structures can be defined by class prefixing. The history, objectives and shortcomings of SIMULA I are covered initially. This is followed by the history and objectives of SIMULA 67. Finally, a detailed description of SIMULA 67 syntax and use (Lamprecht, 1983) is presented.

I. SIMULA I

A. History / Background

SIMULA I or just plain SIMULA is an acronym standing for SIMUlation LAnguage. The language is a true extension of ALGOL 60, containing ALGOL 60 as a subset. It is a language designed to facilitate formal description of the layout and rules of the operation of systems with discrete events or changes of states. SIMULA has extensive list processing facilities and an extended coroutine concept. Simulation is widely used for analysis of a variety of phenomena: e.g. communication networks, traffic flow, production systems and administrative systems. Simulation programs are generally very difficult to write in assembly language or even in a high level langauage like FORTRAN. Therefore a need for a simulation language, one built around a set of basic concepts and allowing for a formal description of the phenomena, is needed to simplify the generation of a program. This language should enable one to observe similarities and differences between systems and allow the user to consider all relevent aspects of the system. This language should also contain an algorithmic language as a subset for the massive amounts of number crunching that are necessary. Finally, the system descriptions should be easy to read and print for the purpose of communication. Acknowledging a need for such a

simulation language, Ole-Johan Dahl and Kristen Nygaard designed and implemented SIMULA at the Norwegian Computing Center under a contract with the UNIVAC division of the Sperry Rand Corporation. Since SIMULA 67 is a direct offshoot of SIMULA, an examination of the development stages of SIMULA is necessary to understand the concepts of SIMULA 67.

SIMULA's history is actually intertwined with that of the Norwegian Computing Center (NCC). The ideas for the language were originated in 1961. In 1962 UNIVAC launched a campaign for their computers, the UNIVAC 1107 and the UNIVAC III. The NCC got a UNIVAC 1107 in 1963 and this resulted in a UNIVAC and NCC software contract. As a part of this contract, the SIMULA compiler was completed in 1965.

B. Development Stages

SIMULA went through four main language stages. The first stage was from mid 1961 to late 1962. In this stage, the initial ideas were based upon a "discrete event network" concept. There were no real specific implementation at this time. The second stage was from late 1962 to late 1963. This stage built on the ideas of the first stage and introduced increased flexibility by the use of the ideas of ALGOL 60. This, however, was somewhat restricting because of the assumption that SIMULA was to be implemented by means of a preprocessor to ALGOL 60. The basic concept of SIMULA

at this time was a system consisting of a finite, fixed number of active components called "stations" and a finite, variable number of passive components named "customers". The station also consisted of a queue part and a service part. The service part had associated with it an operating rule which described the actions of the service part by a sequence of ALGOL statements. The customers did not have an operating rule, but did have variables associated with them called "characteristics". The customers were defined by a real, continuous function called "time" and a function called "position". A customer could be generated by a service part of a station, transferred to the queue part of another station, then to the service part of the station and so on, until it disappeared by not being transferred to another queue part by the service part of some station. These events of the service part of the station were regarded as instantaneous and occuring at discrete points in time. As a result, this class of systems came to be known as a discrete event network.

The third development stage, from late 1963 to early 1964, led to a decision to implement SIMULA through an extension of the UNIVAC's ALGOL 60 compiler, based on a storage management scheme developed by O. J. Dahl. This in turn led to the "process" concept which utilized the new storage scheme. The process concept was intended as an aid for decomposing a discrete event system into components,

which were separately describable. In general, a process has two aspects. It is a data carrier and it executes actions. Declarations used to describe the data, and a sequence of statements, known as the operation rule, described the actions. Unlike procedures, which are dynamically nested, the relationship between processes is a symmetric one. So, the discrete event system was viewed as a collection of processes, whose actions and interactions comprised the behavior of the system. Processes will thus enter and leave the system as results of actions within the system itself. So, the simple network idea was replaced by the more powerful concept of models consisting of processes operating interactively in "quasi-parallel". Processes are user defined transient objects. They are referenced individually. These processes are declared collectively by "activity" declarations. SIMULA would now be implemented by extending the ALGOL compiler and changing parts of the run time system instead of using the ALGOL preprocessor idea. Process queues were now declared explicitly as ordered "sets". In order to increase flexibility, sets were allowed to contain processes of different kinds. The sets were manipulated by "wait" and "include" statements. Quantities that were declared local to the process, called "attributes", were made accessible from the outside by the "extract-select" construct. In other words, the acting process could, by connecting another process, reference the

attributes of the latter as if they were local to the acting one. Thus, the processes were also data carriers, like the customer of the second development stage. Process pointers were introduced as explicit language elements. As a result of much work, all of the major features of SIMULA were now present at this stage.

The final development stage was from early 1964 to late 1964. This stage resulted in the implementation of SIMULA I compiler, which was completed in December 1964.

C. Development Process

SIMULA was regarded as a system description language. It had six main design objectives in 1963. First, there should be a general mathematical structure about which the language should be built. This structure should have a few basic constructs and furnish the user with a standardized approach so that a system can be easily described and defined in terms of these concepts. Second, the language should print out the similarities and differences between various networks. Third, the user should be forced to consider all aspects of the network. Fourth, in addition to being unifying and directing, the language should be general enough to allow the description of different classes of network systems and other systems that can be analyzed by simulation. Therefore, the language should contain a very general dynamic and algebraic language. Fifth, to allow for

communication between users studying networks, the language should be easy to read and to print. Finally, the language should be problem oriented and not computer oriented. This sixth design objective implies increased computer processing.

A year later, in 1964, however, the design objectives were restated. The main differences between the two versions were threefold. First, the term "network" had disappeared from the 1963 "discrete event network system" to which SIMULA was related. It was found that there were many systems that could not be regarded as networks. Thus, the network concept was abandoned and the process concept was now introduced as the basic concept. Second, since a system was now understood as a collection of interactive processes with each process being present in the program execution, the execution of the program existed as a dynamic system within the computer memory. Now, SIMULA was a "dynamic language" rather than being a language built around a general mathematical concept with a few basic constructs. This dynamic language also emphasized a relationship to the block structured language ALGOL. Finally, while the design objective in 1964 still stressed being problem orientation, it now also stressed computer orientation, as it was believed that the success of SIMULA depended on its compile and run time efficiency.

A storage allocation package was designed based on a two-dimensional free area list. In short, each area had a "used" bit that was used to facilitate the combination of neighboring free areas. As a result of this bit, stack allocation was possible and the entire noncontiguous storage of the computer could be utilized. Thus, the search space for dynamic structures was drastically increased bringing about the process concept and quasi-parallel programs. Processes in quasi-parallel execution implied that control could be passed from one process to another as a result of special sequencing statements. The operation of the system was now a sequence of active phases of the processes present in the system. Each process, therefore, had a "reactivation point", which identified the program statement at which control would resume next time the process was activated. The reactivation point created an illusion of a local sequence control which steps sequentially through the statements associated with a process. Thus quasi-parallel processes were parallel in the sense that those processes which currently are inactive can be thought of as "executing" a statement which takes system time. The storage allocation package allowed these sequencing statements to be placed at any program point since their data stacks could grow or shrink independently. Also, processes could be created and destroyed in any order.

SIMULA programs also had to provide programming "security". That is, any erroneous program must be rejected by the compiler, run time checks, or by reasoning based entirely on the language semantics, independent of the implementation. The main objective was to achieve compiler controlled data access. With processes only interacting through nonlocal data, the ALGOL access rules could be applied. All local references could be checked at compile time for validity, except subscripts and parameters which are checked at run time. However, this was not the case when there was a need to obtain access to the contents of an object from outside the object. In other words, the active object would need access to its own data as well as those of the other objects. The automatic storage retrieval mechanism ensures that a computable reference value refers to a process currently in the system. The connection mechanism provided the required compiler control. The user format for this mechanism is as follows

```
INSPECT <reference> WHEN A1 DO S1
                        .
                        .
                        .
                    WHEN An DO Sn
                    OTHERWISE Sn+1
```

where A1,...,An are activities and S1,...,Sn+1 are statements. This forces the user to interrogate class membership of the referenced process. For example, if the process belongs to class Ai the state Si is executed. Si

acts as a "connection" block having the attributes of the "connected" process as its local variables. The connection block contains a stored reference to the connected process. This "connection pointer" prevents accidental deletion of the process while it is connected.

Another form of security dealt with deallocation of storage. This could be done easily by explicitly using a "destroy" statement or going to the process "end". This brings about efficiency and simplified implementation. However, to ensure security, one needed a process referencing technique ensuring that only one pointer could point to a process at any time. Unfortunately, such a scheme was not found. It was finally decided to require that procedures and subblocks be self destructive on exit. The expression

NEW <class> (<actual parameter list>)

gives a value that is a reference to a process. The list of actual parameters provides initial values of attributes of the generated process. Since SIMULA has no delete statement, the process will remain part of the system as long as it can be referenced. The "reference count" of the process is updated each time a reference is stored or deleted. When the reference count becomes zero, the process can no longer be referenced and is deleted. However, this process does not necessarily leave the system when it has terminated its own operations. It may remain as a dead

object, allowing its attributes to be accessible to other processes through the connection mechanism. If memory gets tight, a garbage collection routine deletes reference chains which could not be removed by the reference count mechanism. However, garbage collection is used as a last resort as it was costly in nature to run. However, combining all these together led to possible conflicts with respect to data accessing security. The first conflict that could occur was that a process could outlive its dynamic parent. In other words, a block instance containing the generating expression which gives rise to the process could terminate before the process. As a result, the process may access nonexisting data through its formal parameters. To resolve this, all call by name parameters to processes were disallowed. Another conflict brought about by the deallocation scheme was that a process could outlive its textually enclosing block instance, thereby accessing nonexisting nonlocals. This problem was solved by having all processes be declared by "activity" declations local to a special block known as the "SIMULA block".

SIMULA begin...end

This block is the outermost block or must be embedded in an ALGOL program. The SIMULA block corresponds to the simulation model. On entry to the SIMULA block during execution, the simulation facilities become dynamically

available.

A prominent feature of SIMULA was to be the concept of "process set", along with scanning mechanisms and the "selector expressions" as the only means of process identification. The "process pointer", however, came into being as a result of the selector expressions being inefficient. Efficiency became an important issue. The implementation of the language should be efficient and users should be able to create efficient programs with the language. For example, the built-in mechanisms should have run times independent of the size of the model. In this respect, process referencing became the important issue. An abstract ordered set concept was included as a new data type to be used as a list mechanism for queuing purposes. These ordered sets were implemented as two-way circular lists. Processes were able to be members of any number of sets at the same time by using auxiliary "element" objects to represent a process in different sets. All process references were made indirect by these element objects by providing only "element pointers" in the language. Physically, the process reference was a pointer to an area of memory containing the data local to the process and some additional information defining its current state of execution. A process would remain part of the system as long as it could be referenced through a computable element expression. The element and set concepts served to

facilitate and standardize the manipulation of queues and other linear lists of processes. A set is an ordered sequence of elements. Each element consists of a pointer to the successor element of the set, a pointer to the predecessor element of the set and the pointer to a process. By using defined system procedures, sets can be formed and manipulated. All sets have one dummy element alled the "set head". An empty set therefore consists of only the set head.

The element concept and the technique of reference processes had many desirable properties. First; the ordered sets of processes could be manipulated by means of standard procedures. Next, when a process was referenced through an element in a set, its successor and predecessor in the set were immediately accessible. Also, any given process could be a member of an unlimited number of sets at the same time. And finally, the members of a set could be processes of different classes.

Model simulation time is the time reference used within a simulation model to keep the advancement of time under control in order to allow the computer to simulate concurrent events. The actions of a process are grouped together in active phases, separated by periods of inactivity. Only one process can be actively executing at any one time. An inactive period of a process is caused by a deactivating statement executed by that process. Thus, the current active phase of the process ends and control

leaves the process. A "reactivation point" is held until the time of the next active phase of the process and resumes control at that point. The reactivation point concept allows the user to string together actions occurring at different times into a logical sequence. This active phase of a process is called an "event". Deactivating statements allow for inactive periods of definite or indefinite lengths of time. An event can be scheduled to happen either immediately or at some later time. A process for which an event has been scheduled but not completed has an associated "event notice" representing the event. An event notice contains a reference to the process and a time reference. Implementation of model time scheduling was accomplished by maintaining a list, called the "sequencing set, SQS", of scheduled event notices sorted by time attributes. The SQS was represented by a binary tree which preserved the order of elements with equal time values. This implementation reduced search times but required space for several pointers and other information with each element on the list. To avoid wasting space in processes not on the list, the event notices, as explained previously, are stored on the list. Since each process has only one event notice, the logical significance of the time list is unchanged. Algorithms for removal and insertion of event notices were also implemented. The currently active process is the one which is at the end of the time list. When the current active

phase is completed, the current event notice is deleted. Its successor in the SQS becomes the current event notice, and control enters the associated process at its reactivation point.

A process can be in four possible states. As simulation proceeds, the states of processes will change. A process that is "active" can alter the states of other processes along with its own state. A "suspended" process has an event notice and a reactivation point. This process will start when the event notice becomes the current one. A process that is "passive" has a reactivation point but lacks an event notice. It remains passive until another process changes its state. A "terminated" process reached the end statement of its process definition. This process has no reactivation point or event notice. It can no longer change to any of the other states once terminated. However, a passive or terminated process will remain as it still can be referenced through an element expression. The states of processes are altered by sequencing statements operating on the SQS. A sequencing statement may delete an event notice and/or schedule an event by generating an event notice. In addition, the statement will specify explicitly either the time reference of the event notice, or its position in the SQS. A timing clause specifies the reference of the generated event notice, and this determines its position in the SQS. The event notice is normally placed behind all

others with the same time reference unless otherwise specified.

D. A Detailed SIMULA Example

SIMULA was used to a large extent as a system description language. It was found that the writing of the program or system description almost always led to a better understanding of the system. After the introduction of SIMULA I, many shortcomings were discovered within the language. These shortcomings, which will be discussed in a later section, resulted in the development of SIMULA 67. Below is a complete SIMULA program (See McNeley, 1967) that could be used to represent a system of a grocery store checkout, consisting of two checkers and one hundred customers.

```
Line                    Program
  Ø SIMULA store: BEGIN
  1 ACTIVITY customer(n); VALUE n; REAL n;
  2   BEGIN REAL stime; INTEGER i;
  3     stime:=TIME;
  4     IF n<=6Ø THEN i:=1 ELSE i:=2;
  5     IF EMPTY(queue(i)) THEN ACTIVATE c(i) AT TIME;
  6     INCLUDE(CURRENT,queue(i));
  7     PASSIVATE;
  8     HISTO(h1,h2,TIME-stime,1);
  9     HOLD(Ø.25);
 1Ø     ACTIVATE c(i) AT TIME;
 11     ncus:=ncus+1; REMOVE(FIRST(queue(i)); END;
 12 ACTIVITY clerk(i), VALUE i; INTEGER i;
 13   BEGIN
 14     l1: IF EMPTY(queue(i)) THEN GOTO 12;
 15         INSPECT FIRST(queue(i)) WHEN customer DO
 16           BEGIN HOLD(MAX(Ø.25,Ø.1*n));
 17             ACTIVATE FIRST(queue(i));
 18             PASSIVATE;
```

```
19            GOTO 11; END;
20      12: PASSIVATE;
21          GOTO 11; END;
22 SET queue(1:2); INTEGER ncus; ELEMENT c(1,2);
23 ARRAY h1(1:26), h2(1:25);
24 ncus:=Ø; FOR i:=1 STEP 1 UNTIL 25 DO BEGIN
25 h2(i):=Ø.25+(i-1)*Ø.25; h(i):=Ø; END; h(26):=Ø;
26 c(1):=NEW clerk(1); ACTIVATE c(1) AT TIME;
27 c(2):=NEW clerk(2); ACTIVATE c(2) AT TIME;
28 11: ACTIVATE NEW customer(draw(1:25)) AT TIME;
29 HOLD(EXPON(Ø.25);
30 IF ncus<=1ØØ THEN GOTO 11;
31 HPRINT(h1,h2,1,26,Ø.25,1);
32 END;
```

The program shows reserved words in capital letters. The line numbers are not part of the program but only included for reference purposes. There are two process definitions in this program. The first is "customer". This process contains a parameter "n" which is set outside the process and represents the number of items a particular customer wants to purchase. There is a customer process for every customer in the store. Each customer is described by the same process description but has different values for its attributes "n", "stime" and "i". Lines 3 thru 11 form the process description for customer. Line 3 stores the time when the customer arrives at the checkout. Line 4 determines if the customer has six or less items. This determines at which checkout the customer waits since of the two checkouts in the system, one is for six or less items and the other is for over six items. Line 5 will, if the queue of clerk c(i) is empty, alert that clerk by executing a scheduling statement which schedules that clerk's process to occur at the current point in simulation

time. In lines 6 and 7, the customer is entered on the queue of the clerk and the process of the customer is set to inactive until the clerk checks his items. The time of delay depends on the number of customers already in that clerk's queue. Lines 8 and 9 allow for customers records or histogram to be kept and also delays the customer one quarter unit. After the delay, in line 10, the clerk process is resumed again at line 19. In line 11, the variable "ncus" is global and is used to tally the number of customers already processed. The customer is then removed from the queue and leaves the system.

As stated before, there are only two clerks in the system. Thus, the activity "clerk" will be activated two times. This process contains a parameter "i" which is set outside the process and represents the clerk identification number. Lines 12 thru 21 form the process description for clerk. Line 14 determines if the specific clerk's queue is empty. If it is empty, control shifts to line 20 and the process passivates (becomes passive) otherwise control continues from line 15. Statement 15 extends through line 19. The form "INSPECT P1 WHEN A1 DO" is the way SIMULA establishes which process description the process specified by P1 was created. If it came from the process description specified by A1, the "DO" part of the statement is executed. Lines 16 and 17 cause delays for a fixed time and then the clerk alerts a particular customer process. The customer

process resumes at line 8. The clerk process becomes inactive until the current customer activates it on line 10. The clerk, when reactivated, resumes at line 19. This in turn allows the clerk to check for another customer in the queue. Notice that none of the clerk processes ever terminate because the "end" statement is never reached. This is unlike a customer process which does terminate and leave the system. The clerk process can be termed as a "permanent process", whereas the customer process is a "temporary process". Temporary processes can allow vast amounts of data to pass through the system over a period of time, but only a limited amount of data will be present at any one time.

The main part of the program begins at line 22 with the declaration of set "queue" which is associated with the process clerk. The array "c" is defined containing members that are pointers to the clerk processes for reference purposes. Line 23 defines histogram variables that will be used to record the results of the simulation. Lines 24 and 25 initializes the customer counter and the histogram variables. The two clerks are created and their references stored into c(1) and c(2) on lines 26 and 27. They are activated at the current time. Line 28 creates a customer having one to twenty-five items and is activated at the current time. Lines 29 and 30 cause delays for a period of time to allow for a new customer approximately every quarter

unit of time. Then a check is made to see if all customers were created. Line 31 prints out the histogram information for the simulation run. On line 32 the end of the SIMULA block is reached and the simulation is terminated.

II. SIMULA 67

A. Shortcomings of SIMULA I

As experience with SIMULA increased, a number of shortcomings were found. The element/set concept was found to be rather clumsy as the basic mechanism for list processing. Single process pointers restricted to one set at a time proved by experience to be much easier. The inspect mechanism, which was used for remote attribute accessing, also turned out to be very cumbersome. This led to the idea of record classes. Full Security could be obtained in constructs like "M.C" by compile time reference qualification. The idea of record subclasses turned out to be a reasonably flexible way of run time referencing.

It was felt that SIMULA's simulation facilities were a heavy load to carry for a general-purpose language. The multistack structure worked very well for sequencing, but quasi-parallel sequencing could be used for other applications that did not use the simulated time concept. When writing simulation programs, it was also observed that many processes shared common properties such as data and actions. By somehow preprogramming the common properties, much programming effort could be saved. Recall, as explained previously, that call by name parameters were not allowed for security reasons. As a result, parametrization would not be as flexible for preprogramming common

properties. However, the idea of subclasses being extended to apply to processes could be used.

Another shortcoming of SIMULA existed in its implementation. Whenever the number of process activation records was large, as in most simulation runs, much storage space was wasted. For very large simulations, this led to memory space problems. A new compacting garbage collector was found to be more efficient than the combined reference count/garbage collector that was being used in SIMULA. This new garbage collector could take advantage of active deallocation at exit from procedures and blocks easily by moving the free pointer back whenever the deletion occured at the end of the used memory.

Much work went into the feasibility of the record class construct and how to place it into the language. "Prefixing" was found to be the answer. It was decided that prefixing could be done by using a list structure consisting of a "set head" and a variable number of "links". The various processes could be in effect glued to a link to make each link-process pair one block instance. Each process would be a block instance with two layers. The prefix layer would contain a successor and predecessor and other properties of the two-way list membership. The main layer would contain the attributes of the process. This two layer property of the process must be known at compile time to obtain attribute referencing security and compiler

simplicity. The links are declared separately without any information about the other process classes which used link instances as a prefix layer. Since the processes of these other process classes were both links and more, the class is indicated by prefixing their declarations with the process class identifier, namely, "link". These process classes would then be "subclasses" of "link". Prefixing leads to multiple prefixing. This in turn can be used to establish hierarchies of process classes. The concatenation of a sequence of prefixes with a main part could also be applied to the action part of a process class.

The class concept led to a completely new language approach. SIMULA I's shortcomings brought about SIMULA 67 which would have the following points. First, the new general programming language would be designed in terms of being an improved SIMULA I. The basic concept would be classes of objects with the prefix feature and subclass concept included. Finally, direct and qualified references would be introduced.

B. Development

Development began with the unification of the old process like objects and the new concept of self-initializing data/procedure objects. Along with this began the removal of the model time or simulation time concept. The term "object" now came about since the term "process"

really could not be applied to the new concept. This object would be generated like a function procedure by being invoked by the evaluation of a generating expression. The object may then set its own local variables as necessary. The control would return to the generating expression carrying back a reference to the object as the function value by either reaching the "end" of the object or as a result of the "detach" operation. If an "end" was found, the object terminates and no further actions of the object can be executed. On the other hand, a "detach" allows the object to become a "detached object" and be capable of functioning as a "coroutine". The coroutine call "resume (<object reference>)" would make control leave the active object, leaving a corresponding reactivation point at the end of the resume statement, and enter the reference object at its reactivation point.

The declaration given to a class of objects is called "class". The idea of class prefixing and concatenation made it possible to define classes primarily intended to be used as prefixes.

Circular list processing, similar to sets in SIMULA I, were described by means of a class heirarchy for list elements, "class link", and list head, "class list". These both had forward and backward pointers contained in a common prefix part. This meant that any class prefixed by "link" could have objects that could go in and out of circular

lists. Procedures such as "into" and "out" declared within the class prefix part, together with the list pointers, make insertion and deletion possible.

The concatenation mechanism was slightly modified in order that the process concept as a prefix class could be used. Originally, the operation rule of the concatenated class contained the operation rule of the prefix class followed by the main part. Now, for a process object, predefined actions must exist at the front and at the end of the operation rule. Thus, the prefix class had an operation rule of initial actions and final actions split by the symbol "inner". This prefix class was named "process". The term "process class" now was used instead of the "activity" of SIMULA I.

All of the sequencing statement procedures of SIMULA I could be implemented by using procedures that worked on the SQS, the sequencing set. Terminated objects could be removed from the SQS and control passed to the successor object. The only problems that remained were the placement of the SQS pointer and the representation of the main program of the simulation model, which in SIMULA I was accomplished by the SIMULA block. The problems were solved by taking the prefix classes, procedures, and SQS pointer and putting them into a big class named SIMULA. The initial actions of this prefix class was to initialize the SQS which contained the main program actually disguised as a process

object. What was important here was the fact that an instance of a prefixed block is a detached object by definition. This meant that the main program could function as a coroutine in quasi-parallel with its local objects. In June of 1967, the "SIMULA class" was reorganized as a two level hierarchy,

CLASS SIMSET

and,

SIMSET CLASS SIMULATION

This now allowed circular list handling for purposes other than simulation.

Even though the class/subclass facility could be used to define general object classes and specialized subclasses by declaring additional properties, adding details to the operation rules could not be done. As stated before, call by name procedure parameters, which could solve this problem, could not be used because of allocation and security problems. A "virtual" quantity concept, where the actual parameters would have to be declared in the object itself but at a deeper subclass level than that of the virtual specification, was adopted. A generalized object could now be defined whose behavior pattern could be left unspecified in the prefix class body. Different subclasses could then contain different actual parameter declarations.

In 1967, another development began to take shape. String handling and input/output facilities were based on

classes and a new type "character". The class "string descriptor" contains a character array. The class "string" identifies a substring of a string object and a scan pointer for sequential access. Both classes contain various operators declared as procedures. These constructs provided much flexibility but also run time data structure and syntactic overhead. The string type was later changed to "text" by name. A text could be thought of as either a string descriptor ("text reference") or a character sequence ("text value"). A new notation was designed to distinquish them. The operators ":-", "==", and "=/=" were chosen for reference assignment, reference equality and reference inequality. These signs were also applied to object references as well. Input/Output was designed by using a hierarchy of classes corresponding to different kinds of files.

C. Implementation and Standardization

SIMULA I was originally a system description and simulation language, not a general-purpose programming language. It was mainly implemented for the UNIVAC 1100 computer. SIMULA 67, however, was to be a general programming language and as a result be made available on most major computer systems. The Norwegian Computer Center came on hard times in 1967, and became restricted to new large long-range projects. However, four people were

allocated to the SIMULA 67 implementation. But other resources were not made available since arguments against SIMULA 67 still existed. It was felt that SIMULA 67 would not be very profitable to the NCC. The NCC felt that a modern, commercial compiler would require a substantial investment to become profitable. The NCC was not willing to put out large amounts of money for SIMULA 67. But, noting the reputation of SIMULA I and the fact that SIMULA 67 was to be linked to ALGOL 60, along with the importance of simulation, the implementation started.

Top priorities were given to implementaion for Control Data, IBM and UNIVAC computers. Compilation and run time speeds had to be comparable with ALGOL 60 compilers. This, coupled with documentation and educational material would make SIMULA 67 a high standard language. ALGOL 60 was contained as a subset of SIMULA 67 with only minor modifications. The name SIMULA 67 was agreed upon with some reluctance due to the feeling that this language would be considered as a true simulation language. It was feared that it may slow down the language's acceptance as a general-purpose language. However, the name was accepted due to the fact that it was a new improved version of SIMULA I that could be used for simulation.

As of 1976 there were eight different compilers for SIMULA 67. Implementations existed for the UNIVAC 1100 series, CDC 3000, 6000 and Cyber 70 series, IBM 360/370

series along with the DEC system-10 series. Translation programs to transfer SIMULA 67 programs from one implementation to another do exist. It has been found that SIMULA 67 programs were easy to move from one computer to another. The fact that SIMULA 67 allows for no undefined elements allowed the SIMULA 67 language itself to cause few problems. The main problem with moving SIMULA 67 have been outside the language. A few of these problems include hardware differences between computers and also operating system differences mostly in file handling. Also, moving a program from a batch to an interactive environment caused problems.

There are a few incompatabilities that exist between SIMULA 67 systems. First, hardware representation was not considered when SIMULA 67 was defined. What this means is that on some SIMULA 67 systems reserved words are used, and on other systems markers around key operator words are used. For example, on one system "IF" is a reserved word and on another system this may be denoted by "'IF'". Even though translator programs can be used to amend the notations, trouble would have been saved if the hardware representations would have been designed with the language design. Another incompatibility exits because different operating systems handle files differently. Finally, different word lengths on different systems leads to precision problems for real variables.

Standardizing SIMULA 67 so that programs can be easily transported has been difficult due to the desire to add new features, REPEAT-UNTIL for example. Another reason stems from the desire to solve certain problems in a better way then the initial definition of SIMULA 67 allowed them to be solved. All in all, SIMULA 67 was standardized before and during the first implementation. This, on the whole, gave good compatability between the systems and allowed the SIMULA 67 standardization to be more successful than other standardization efforts. The main reason for this is that the SIMULA 67 language is fully defined and does not, as many other languages do, contain undefined constructs.

D. Simulation Language Objectives

In this section, simulation languages in general will be covered. Simulation, in a broad sense, could be defined as a technique of representing a dynamic system by a model in order to gain information about the system through experiments with the model. Digital simulation is widely used as a tool for studying traffic flow, production systems, transportation and communication networks, among others. The simulation language therefore serves the following purposes. First, it aids the analyst in building a model by presenting a conceptual framework for identifying and describing the system components. Next, it provides a notation for this description of the dynamic model. Finally,

it serves as a programming aid, making changes easy to modify.

There are two different approaches used in developing a simulation language. The "continuous approach" is mainly accomplished using analog computers. But since digital computers are discrete devices, continuous changes in the physical system can be represented by a series of discrete changes in the model. This is called a "discrete approach" and such a model is called a "discrete event model". In contrast to the technique of representing the system as a whole by a set of differential equations, the individual events of a discrete model are specified in great detail. Many discrete simulation languages, as a result, are general-purpose algorithmic languages.

Simulation languages also provide concepts and programming facilities not found in ordinary general-purpose programming languages. First, simulation languages enable concurrency of processes by introducing a system time concept used for ordering events. Usually, systems are very large containing vast amounts of data. Dynamic storage allocation of data is a common feature. Components, thus, enter and leave the system, only those currently present are represented within the computer. Many dynamic systems are concerned with motion and flow which means that the configuration of the system changes with time. Therefore, all simulation languages provide some form of list

processing. Interdependence is also present. For example, conditions for given events to occur may be extremely complex on account of interdependence between system components. Most simulation languages, as a result, have general-purpose logic capabilities including set concepts and predicate calculus facilities. Algorithms are present for the generation of random numbers according to various distributions. Statistical analysis is very important in simulation. The consecutive changes of state in a model represent the complete history and outcome of the experiment. In order to get meaningful results, individual observations of selected variables need to be analyzed statistically. Built-in functions to average, histograms and others are standard in simulation languages. Finally, continuous phenomena are in principle represented by a series of discrete changes. Most discrete event languages provide no aids for treating continuous changes.

Simulation languages involve systems in which interrelated processes interact in time. Processes are modeled by a sequence of discrete "events", each of which is assumed to occur instantaneously in the time scale of the system. The effect of an event is to change the "state" of the system. The total effect of the process is the sum of the effects of the sequence by which the system is characterized. A scheduling algorithm determines the event with the earliest time from a list of events which have been

scheduled and causes execution of that event. "Exogenous events" are scheduled by a mechanism outside the system being simulated, while "endogenous events" are scheduled during the execution of other events. The information structure on which events operate are referred to as "entities". An entity forms a single unit with respect to creation or deletion but may have a number of data fields of different value types. The entities manipulated by event subroutines of a simulation language include both data entities, which specify data attributes of the process and event notice entities, which specify information about events which have been scheduled for execution at a point in system time not yet executed. When execution of the event completes, the scheduler determines the next event to be executed. Simulation algorithms allow the user to have explicit control over the order in which simultaneous events are to be scheduled. Specifics of the scheduling mechanisms, quasi-parallel processing and the simulation algorithms will be covered in greater depth in the section on SIMULA 67 syntax.

III. SIMULA 67 Constructs and Syntax

A. Statements

As is standard in most languages, SIMULA 67 includes the numerical data types of REAL and INTEGER. Also included is the BOOLEAN data type which takes on the value of "true" or "false". To declare a variable of any one of these types, the following example shows the syntax that is used.

```
REAL x,y;
INTEGER z;
BOOLEAN found;
```

This example accomplishes the following task. First, storage locations are set aside with the names "x", "y", "z", and "found". The type of the variables "x" and "y" is fixed as real and "z" as integer. The variable "found" is fixed as a boolean. Finally, variables "x", "y" and "z" will have initial values of zero and the variable "found" will be set as "false". Every variable that is used must be declared before being used for the first time. The declarations appear at the top of a block. Variable names must begin with a letter and may be followed by letters and/or digits depending on the specific compiler used.

The operations that can be performed on numerical variables and constants are addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (**). Exponentiation has the highest priority of the operators.

Multiplication and division share the next rank, followed by addition and subtraction sharing the lowest rank. The typical rules for evaluation of an arithmetic expression hold. That is, left to right evaluation depending on operator priority. If both operands in an expression are integer variables, the result is integer, however, if one of them is a real variable, the result is real. If the operator is division (/), the result is real in any case. For integer division, an operator of double slashes (//) can be used. If the operator is exponentiation, the result is real in all cases.

An assignment statement takes a typical form of variable followed by the ":=" assigning mark, followed by some expression. For example,

```
x := y + x;
```

adds the integer variable "x" to "y" and stores the result back into "x". Notice also that each assignment statement, as in the variable declarations, is followed by a semicolon. The semicolon acts as a separator between statements.

Labels on statements are also possible in SIMULA 67 programs. A label name has the same restrictions as a variable name. A label name must be different from all other names assigned to variables. The label is separated by a colon (:) from additional labels, which may follow, and from the succeeding statement. The following example,

```
loop: x := x + 1;
      GOTO loop;
```

shows a label called "loop". It also shows an "unconditional" jump by using a "goto" statement. As a result of this "goto" statement, the piece of code will be executed endlessly resulting in an infinite loop. As in languages such as PASCAL, the "goto" statement is a basic part of the language, however, due to the nature of the language being a structured programming language, the "goto" is seldom used. Other "conditional" jumps are used instead of the "goto" to allow for a perfectly structured program.

Conditional statements take the form of

```
IF b THEN s1 ELSE s2;
```

where "b" is a boolean expression. If this expression has the value of "true", the statement "s1" is executed and the statement "s2" is skipped. On the other hand, if the boolean expression has the value "false", the statement "s2" is executed and the statement "s1" is skipped. In both cases, the program is continued with the statement following the conditional one. The statements "s1" and "s2" are restricted to being only a single statement in each case. However, it is possible to execute a series of statements in place of the single statement "s1" or "s2" by forming a "compound statement". A compound statement is formed by joining a series of statements together as a unit by placing

the word BEGIN before the first statement of the group and the word END after the last statement of the group. In general, at any place where a single statement is permitted, a compound statement is also permitted. The IF statement can also be used without the ELSE clause in the following manner.

```
IF b THEN s1;
```

In this case, statement "s1" is executed if the boolean expression "b" is "true" otherwise the program control will continue with the next statement following the conditional one.

Loops can be handled in another way besides using GOTO statements. The loop can be carried out using a FOR statement.

```
FOR c:="list" DO s;
```

Here, "list" can be replaced by elements separated from one another by commas. Each element can have one of three forms:

```
l STEP i UNTIL u
e WHILE b
```

or,

```
e
```

The control is performed for each element of the FOR statement, one after the other. The first form,

```
FOR c:=l STEP i UNTIL u DO s;
```

uses the "c" as a control variable, the "l" as the starting value, the "u" as the upper bound and the "i" as the increment that repeats statement "s". When this form of the FOR statement is encountered, the control variable "c" is assigned the lower bound "l". This control variable is tested to see that it has not exceeded the upper bound "u". If is has not, the statement "s" is executed. However, if the control variable exceeds the upper bound, the program is continued from the statement following the FOR statement. If the control variable was not greater than the upper bound, the control variable is then incremented by the "i" value and execution loops to check and see if "s" should be executed again. As long as "c" is not larger than "u", the statement "s" gets executed and "c" gets incremented by "i" in a loop. The second form of the FOR statement,

```
FOR c:=e WHILE b DO s;
```

works as follows. The control variable "c" is given a value of some expression "e". The boolean "b" is then tested. If "true", statement "s" is executed and the loop continues. If "false", the loop ends and the program is continued with the statement following the FOR statement. The last form of the FOR statement is as follows.

```
FOR c:=e DO s;
```

The value of the arithmetic expression "e" is assigned to control variable "c". The statement "s" is then executed. Finally, the statement following the FOR statement is executed. As a result, this form of the FOR statement is not a loop as the other forms are, as the "s" statement is executed only once. The following is an example to end the discussion on FOR statements.

```
FOR r:=6, 9 STEP 2 UNTIL 17, 20 DO
  BEGIN
    t := 2 * r;
    g := 3 * r;
  END
```

The loop will calculate values for "t" and "g" for the values of "r" at 6 (first element), 9, 11, 13, 15 and 17 (second element), and 20 (third element).

Another way to generate a loop is by using a WHILE statement.

```
WHILE b DO s;
```

As long as the boolean expression "b" is "true", the statement "s" will be executed. The WHILE statement is not part of the SIMULA standard, however, most compilers accept it.

Attention will now be turned to relational and logical operators. The relational operators are less than (<), less than or equal (<=), greater than (>), greater than or equal (>=), equal (=) and not equal (¬=). The result of two

arithmetic expressions separated by a relational operator is either "true" or "false", as is typical of most other languages. The logical operators, in order of highest to lowest priority, are negation (NOT), logical AND (AND), logical OR (OR), implication (IMP) and equivalence (EQV). A boolean expression is evaluated on the basis of priorities of these operators. As an example, the assignment statement,

```
m := r<=5 AND r>Ø;
```

will set the boolean variable "m" to be "true" if the integer value "r" is greater than zero and less than or equal to five.

A vector variable can be stored in memory by using the ARRAY declaration. This declaration will reserve an area of locations in memory and attach to it a name. These places will all have the same specified type. The declarations also fixes the bounds for the indexes. The following,

```
INTEGER ARRAY a,b (6:2Ø);
REAL ARRAY c (1:5,1Ø:15);
```

represents the declaration of two vectors "a" and "b" of fifteen elements each, both with indexes from 6 to 2Ø. These two vectors may hold only integer values. The second declaration sets up a two-dimensional array , "c", with the first index varying from 1 to 5 and the second index from 1Ø to 15. The "c" array has thirty elements and only holds

real values. Boolean arrays can also be declared.

B. Input/Output

Input and output in SIMULA 67 work according to a system of card images. For input, the card image of eighty places is transferred to a buffer, called SYSIN.IMAGE, with eighty locations by the following command.

```
INIMAGE;
```

Therefore, if input is desired, the INIMAGE command must be given in order to place the external data into the internal buffer. An integer value can then be read from this buffer by using the command,

```
v := ININT;
```

This puts an integer value into the integer variable "v". In the same respect, a real value can be read from the buffer by using the command,

```
x := INREAL;
```

This will put a real value into the real variable "x". In SIMULA 67, a digit must follow the decimal in a real number. For example, "3." is not allowed as a real value, but "3.Ø" is. Both ININT and INREAL are system function names.

Card images can contain several values. To accomplish this, a position indicator or pointer, called

SYSIN.POSITION, is attached to the buffer. When the command INIMAGE is executed, the position pointer is set to the beginning of the buffer. On each reading from the buffer, the position pointer is also changed. Between each number on the card image there has to be at least one space. Thus, after a number is read, the position pointer points to the space after the number.

The position of the buffer pointer can be set by the user by means of the command,

```
SYSIN.SETPOS(n);
```

where "n" is the position location on the buffer. The pointer may be moved forward or backward with regard to its present position. This will allow for data or whole card images to be read any number of times.

If a program is needed to process an unknown number of data card images, SIMULA 67, as many other languages also do, provides a boolean name ENDFILE to indicate whether the end-of-file has already been read or not. ENDFILE will remain "false" as long as at least one card image can be moved into the input buffer by the statement INIMAGE. Therefore, when the end-of-file is reached, ENDFILE will become "true".

Output also works by using a buffer. This buffer, called SYSOUT.IMAGE, handles 132 characters. The buffer is printed to output by using the statement,

```
OUTIMAGE;
```

At the beginning of the program and after each OUTIMAGE, a fresh buffer is set to blanks and the buffer position pointer, called SYSOUT.POSITION, is set to the beginning of the buffer. Values can be sent to the buffer by using one of the following commands,

```
OUTINT(v,w);
OUTFIX(v,a,w);
```

or,

```
OUTREAL(v,a,w);
```

For the OUTINT command, the "v" stands for the variable to be printed and "w" is the width of the field that the variable is to be printed in. If the width that is chosen is too small for the output of the number, the value is not output. However, a row of asterisks is output in the field instead to show that the field width was too small. For the OUTFIX and OUTREAL commands, the "v" again stands for the variable to be printed and the "w" again for the field width. The "a" stands for the number of digits behind the decimal point that is wanted. OUTFIX transfers the variable as a fixed-point number into the output buffer. OUTREAL transfers a variable as a floating-point number to the output buffer. Again, if the width is too small, the field will be output with asterisks. As with the input buffer position pointer, the output position pointer can also be positioned by the user by the command,

```
SYSOUT.SETPOS(n)
```

Here, "n" is a number between 1 and 132.

Text may also be output by using the command,

```
OUTTEXT("This is text");
```

The words "This is text", minus the quotes, will be sent to the output buffer. The text to be output must appear between two quotation marks.

C. Text

A text, in SIMULA 67, is treated as a three level instance. The first level, "text reference", refers to the second level, a "text descriptor". This text descriptor contains the address "a" of the area of the text, the text length "l", a pointer "p" to the next character of the text and the displacement "d". In the third level, the "text field", the contents of the text is stored and a place "m" held by a reference to the text descriptor.

The declaration,

```
TEXT x,y,z;
```

enables the address of the text descriptors in "x", "y" and "z". At this point, the text reference is initialized to the name "NOTEXT". The text descriptor has the value zero for the items "l", "p" and "d". The text field is empty except for "m" pointing back to the text descriptor.

Although "x", "y" and "z" now have been initialized, along with a reference to an empty text, no instance is available to store a text. This is done by using the statement BLANKS. The relation between the text variable declared and the instance created is produced by the reference assignment,

```
x :- BLANKS(15);
```

This creates a text instance of a field up to fifteen characters. The number fifteen can be replaced by any number depending on what is needed. This text field is filled with blanks and variable "x" refers to it. To assign a text to the text field of "x", the assignment character is used.

```
x := "This is a text";
```

This statement transfers the text on the right side, minus the quotes, to the area of the text instance to which "x" refers. Since the length of this text is only fourteen characters, the remaining place of the text area is filled with a blank.

Another way of setting a text instance is by using the COPY command,

```
y :- COPY("This is also text");
```

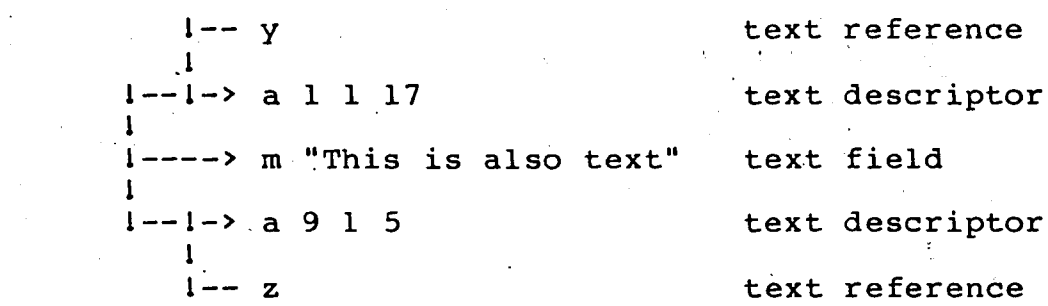
This creates a text instance that is referred to by "y" and

also transfers the text immediately to the text area of the instance. This text area will have a length of seventeen since the text string contains seventeen characters.

A reference to part of a text instance can be assigned to another variable. The following example explains the command that is used to accomplish this.

```
z :- y.SUB(9,4);
```

This command will assign to variable "z" part of the text instance "y". This is done by placing a decimal point behind the variable "y" and following it with the word SUB. In the parentheses, the displacement from which the subtext is to start is nine, and the length of the subtext is four. This means that the text area of "z" contains the string "also". To further explain, a graph of the situation is used.



It is seen by this graph that the text descriptor of "y" and "z" both have address "a", meaning that both point to the same text field. However, the variable "y" is a reference to the whole text since the displacement field is one, the

pointer character is one, and the length field is seventeen in the text descriptor. On the other hand, the text description of "z" shows that "z" is a reference to only a subtext of the text field. The displacement field of "z" is nine, the pointer character is one and the length field is five.

To take this a step further, if a text value is assigned to "z" as follows,

```
z := "more";
```

the text field of "y" is also altered since "z" refers to a subtext of "y". This will then result in changing the text field to "This is more text". For each text field, any number of subtexts can be defined. These subtexts may overlap. Subtexts must not exceed the boundaries of the original text.

To reiterate, a text reference assignment assigns a variable to a text description that points to a text field. This is expressed by the ":-" symbol. A text value assignment assigns new contents to a text field by using the symbol ":=".

The type CHARACTER can be used to store characters in variable locations. The declaration,

```
CHARACTER a,b;
```

will provide two single byte storage places with the names

"a" and "b" with the type CHARACTER. These places will be initialized with nonprintable "ØØ". A value can be assigned to a variable by using as assignment statement.

```
b := '*';
```

This will store an asterisk in the variable "b". Notice that the character is surrounded by single, not double, quotes. Character arrays can also be declared by using

```
CHARACTER ARRAY
```

along with the variable name and boundary limits. The function CHAR(n), where "n" is an integer, will return a character corresponding to the bit pattern of the given integer. The opposite is accomplished by the function RANK(c). This will return an integer corresponding to the bit pattern of the character "c".

Other predefined functions besides CHAR and RANK are present in SIMULA 67. SYSIN.MORE is a function that returns "true" if the position indicator of the input buffer is not greater than the length of the buffer. Therefore, if all the characters of the buffer have been read and no more can be transmitted, the SYSIN.MORE will be "false". The function LETTER(c) will be "true" if "c" is a capital letter and "false" otherwise. The function DIGIT(c) will be "true" if "c" is a digit and "false" otherwise. If "a" is declared as a character, a character can be read from the input

buffer, SYSIN.IMAGE, by,

```
a := INCHAR;
```

and a character can be sent to the output buffer, SYSOUT.IMAGE, by

```
OUTCHAR(a);
```

If "x" is a text variable with a text field of length eighty characters, the contents of a card image can be transferred to "x" by,

```
INIMAGE;
x := INTEXT(80);
```

First, the card image is sent to the input buffer and then the entire text of eighty characters is transferred to the text field referenced by "x". In general, any number of characters can be transferred from the input buffer by the INTEXT function.

When reading values from the input buffer, the names INCHAR, ININT, INREAL, and INTEXT can be used. Values can also be read from a text variable or a subtext. For example, suppose "x" is a text variable. The statement,

```
y := x.GETCHAR;
```

with "y" being a character variable, will transmit a character from which the position indicator of "x" is pointing, to "y". The position indicator is then

incremented. The statements

```
y := x.GETINT;
```

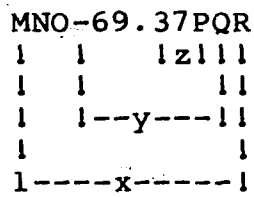
and,

```
y := x.GETREAL;
```

with "y" being an integer in the first case and a real in the second case, will start at the first position of the text until the corresponding value is found. If the first character is not a digit, an error will occur. The position pointer will be placed following the found number. The following example,

```
BEGIN
  TEXT x,y,z;
  CHARACTER c;
  INTEGER i;
  REAL r;
  x :- COPY("MNO-69.37PQR");
  y :- x.SUB(4,8);
  z :- x.SUB(8,3);
  i := z.GETINT;
  r := y.GETREAL;
  c := x.GETCHAR;
END
```

can be explained as follows. The text field is shown graphically below.



The statement "i:=z.GETINT" will set "i" to be 37, while the

statement "r:=y.GETREAL" will set "r" to be -67.37 and the statement "c:=x.GETCHAR" will set "c" to be the letter M. To note, for example, the statement "i:=z.GETINT" could also have been written as "i:=x.SUB(8,3).GETINT".

The position indicator of a text variable can be set by the statement,

```
i := x.POS;
```

which puts the value of the position indicator of the text variable "x" into the integer variable "i". The value can be changed by using the command

```
x.SETPOS(n)
```

where "n" is a value between 1 and the length of the text. The length of the text can be put into an integer variable "i" by the command

```
i := x.LENGTH;
```

Values can also be written into text fields by other means than from the input buffer.

```
x.PUTCHAR(c);
```

The above statement will put a character "c" into the text variable "x" at the position pointed to by the position pointer. The pointer then gets incremented. The following statements will put numbers into a text field "x".

```
x.PUTINT(i);
x.PUTREAL(r,a);
```

and,

```
x.PUTFIX(r,a);
```

Here, "i" is an integer variable, "r" is a real variable and "a" is the amount of digits after the decimal point.

The typical relational operators can be used for characters. Besides the comparison of text variables with the relational operators, there are also ways to test whether a text variable refers to the text field as another text variable. The operator "==" is used to test the equality of references, while "=/=" is used to test for inequality. Therefore, if two text variables "x" and "y" reference the same text field, the comparison,

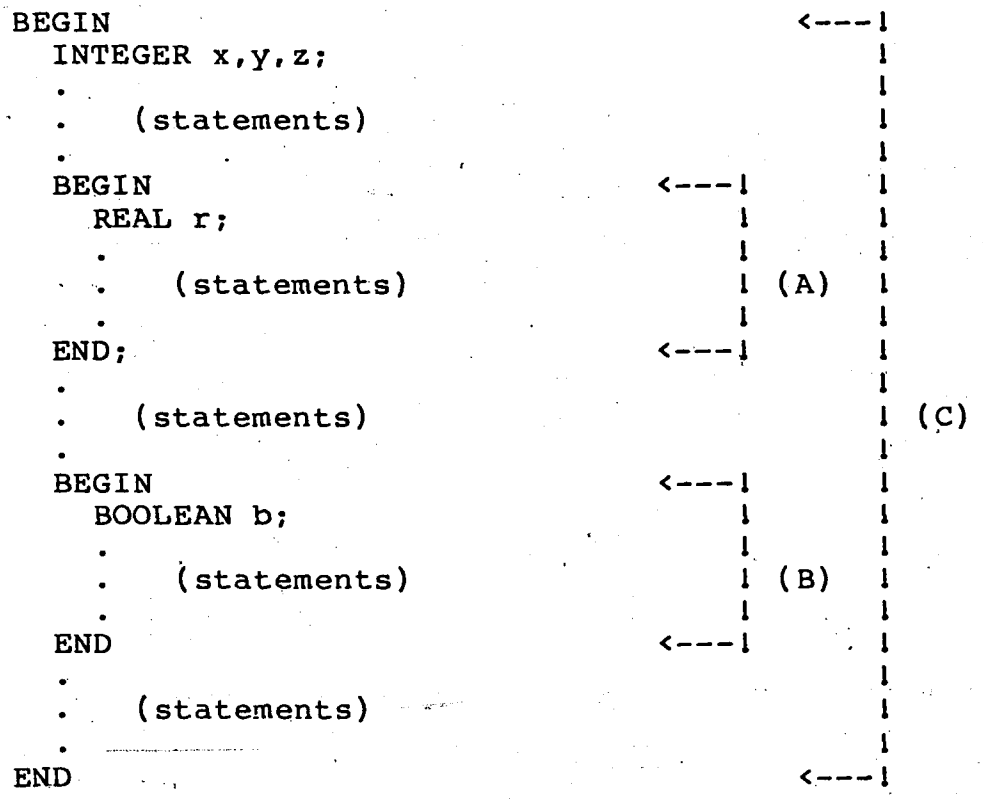
```
x==y
```

will result in the value "true".

D. Program structure / Blocks / Subprograms

A SIMULA 67 program is composed of one large block consisting of internal blocks. A block is a unit that contains declarations and statements surrounded by the keywords BEGIN and END. A block can appear wherever a typical single statement can be placed. As seen previously, a compound statement is actually a form of a block which has no declaration statements. Blocks can be contained within other blocks. The variables that are declared inside the

block are said to be "local" to that block. All variables that were declared outside the block are said to be "global" to the inner block. The following consists of three blocks, which graphically are labelled A, B and C.



It is seen that blocks A and B are inside block C. Block A has variable "r" declared. After entrance into the BEGIN, this variable will be created somewhere in storage. This variable is local to block A. As soon as the END statement is reached, variable "r" will no longer exist. Block B has variable "b" declared. Again, after entrance into the BEGIN of this block, the variable will be created somewhere in storage. This variable is local to block B. Again the variable will disappear upon ending of the block. Block C

has three variables, "x", "y" and "z", declared. These variables are local to block C. Since blocks A and B are contained in block C, these three variables are global to blocks A and B. As a result, blocks A and B have access to these variables and can use them in any way. On exit of block C by the END statement, these three variables will disappear.

The definition of blocks allow for what is called "dynamic array declaration". This is accomplished by setting the dimension of an array in an outer block and working with the array in an inner block. The following is an example of dynamic array declaration.

```
BEGIN
  INTEGER i;
  INIMAGE;
  i := ININT;
  BEGIN
    CHARACTER ARRAY ch(1:i);
    .
    .   (statements)
    .
  END;
END
```

Here, an integer "i" is read in from the input in the outer block. This means that "i" may change each time this piece of code is executed. The inner block then creates a character array "ch" of "i" elements. Therefore, since the "i" can change, the array may have a different number of elements at different times. This is a dynamic array since the size of the array is not pre-set before the compilation

of the program.

Suppose now, that a particular task is to be executed a number of times in various parts of a program. For example, the average of twelve numbers is to be found at ten different locations within the program. A block, which contains the code to do the averaging, could be placed at each of the ten locations within the program. This tends to make the program very redundant and large. A more efficient way to accomplish this task is to create a subprogram that contains the averaging code. The subprogram appears only once and can be called from any point of the program. However, before it is called, it must be declared. This declaration, as was the case with variable declarations, must appear at the beginning of the block, and is usually at the beginning of the program so that the entire program may have access to the subprogram. There are two types of subprograms. The first is called a "function procedure". This type returns a value to the calling statement. The second type is called a "proper procedure". This type does not return a value to the calling statement as the function procedure does, but, performs a specified task.

The declaration of a function procedure has the following structure:

```
type PROCEDURE procname (formal);    ---¦  procedure
  formal parameter specification     ---¦    head
BEGIN                                ---¦
  variable declarations                 ¦
  statements                            ¦  procedure
  procname := ...;                      ¦    body
END;                                 ---¦
```

The procedure head contains the type of procedure. The type can be BOOLEAN, CHARACTER, INTEGER, REAL or TEXT. The keyword PROCEDURE is followed by a procedure name. This is then followed by a list of formal parameters. The formal parameters are substitutes for the actual parameters of the calling statements, replacing them when the subprogram is called. The formal parameters are then specified by type in the next line. The procedure body lies between the BEGIN and END. It contains declarations of any other variables that are used and also all the statements needed to perform the specific desired task. Finally, a statement returning the value wanted to the calling statement must be present, as indicated by

```
procname := ...;
```

The following example is a function procedure named "average" that will compute the average of a twelve element real array. Since the value returned is a real value, the subprogram is a real function procedure. The only formal

parameter is the array. Note, that there is no boundary given in the formal parameter specification since it will have the same boundaries as the actual parameter.

```
REAL PROCEDURE average(arr);
  REAL ARRAY arr;
BEGIN
  INTEGER i, sum;
  sum := 0;
  FOR i:=1 TO 12 DO
    sum := sum + i;
  average := sum / 12;
END;
```

In this example, the call to the subprogram is carried out by the statement,

```
result := average(avgarr);
```

where "result" is a real variable and "avgarr" is a real array of at least twelve elements. The value returned from the subprogram "average" will thus be placed in the variable "result".

A "proper procedure" does not return a single value but executes statements with the intent of using actual parameters to perform a specified task. Therefore, a type does not have to be defined for a proper procedure. The main structure of a proper procedure is the same as a function procedure accept no type is defined in the first line.

```
PROCEDURE procname(formal);
```

Also, there must be no line "procname := ..." present in the procedure body.

There are three ways that actual parameters can be transferred. They are "call by value", "call by name", and "call by reference". Keywords are VALUE, for "call by value", and NAME, for "call by name". Call by reference does not have a keyword, but is automatically used for TEXT and ARRAY types.

A proper procedure is shown below that will swap the contents of two integer variables.

```
PROCEDURE swap(x,y);
  INTEGER x,y;
BEGIN
  INTEGER temp;
  temp := x;
  x := y;
  y := temp;
END;
```

In this example, the call of the subprogram is carried out by the statement,

```
swap(a,b);
```

where "a" and "b" are integer values. After execution of this statement, the contents of "a" and "b" will be switched.

SIMULA 67 also has the capability of allowing a procedure to call itself. This type of procedure is called a "recursive procedure". Recursive programming is a subject all to itself and will not be covered in this paper.

Finally, predefined subprograms are also supplied for SIMULA 67. They can be called as if they had been declared as procedures at the beginning of the program. Many of these deal with mathematical functions like ABS(x) for absolute value, SIN(x) for sine of an angle, and SQRT(x) for square root of a number. Other predefined subprograms deal with handling texts and characters. These include functions that were already explained, like, BLANKS(n), COPY(n), t.LENGTH, and ININT.

In conclusion, blocks and procedures have the following useful properties. First, a block defines an entity that has properties and performs actions. A block where only local quantities are referenced is a completely contained program component. A block is itself a statement, which is a syntactic category of the language. Finally, 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. As a result, storage allocation cannot be administered as a simple stack. A garbage collector, using a scan-mark operation, is required to detect and reclaim those areas of storage which can no longer be referenced by the running program. Such a procedure which is capable of giving rise to block instances which survive its call is known as a "class" and will be addressed in the next section.

E. Class Construct

The notion of "class" and "object" can be traced to the notions of "block" and "block instance" in ALGOL 60. A "block" in ALGOL contains the description of a data structure and associated algorithms. When the block is executed, a dynamic "block instance" containing the local variables of the block along with information for dynamic linkage of this block to other blocks, is generated. By using "class", it is possible to generate multiple "objects" of the same class. Each object generated is a class instance. An object may suspend its execution and start the execution of a different object. When a class contains no algorithms, the class is a "record class" with its objects being "records". Objects are really records to which algorithms are associated. Different objects can exist in memory at the same time. These objects may be of the same or different classes.

A class is defined and used in three phases. The first phase is the description or declaration of the class. This phase declares all the variables with their types that are needed for the class bearing the name provided. Along with the variables, a list of the instructions of the class are defined. The second phase is the realization or incarnation of the class. In this phase, a real copy of the class is created in working storage. A reference to the incarnation is also established and stored in a variable. The final

phase is the actual use of the incarnation of the class. Here, the elements of the incarnation get values which can be used again later. A class declaration has the following form,

```
CLASS name(formal);                          <---!  class
  formal parameter specifications            <---!  head
BEGIN                                        <---!
  declarations                                    !
  initial statements                              !  class
  INNER;                                          !  body
  final statements                                !
END;                                         <---!
```

The class head consists of the keyword CLASS followed by the name of the class and any parameters which this class is to use. Variable declarations, along with subprogram declarations, can be made is the class body. First, an instance of the class must be created; only then can a declared procedure be called by a reference variable. The class body also consists of initial statements or operations and final operations. Any kind of statements are allowed. These can include assignments, input/output, subprogram calls, or even creation of new class instances. The symbol INNER represents a dummy instruction acting as a separator between both sets of operations. More on INNER will be discussed later. The parameters or the variable declared are attributes of the class and also attributes of any object of that class. Also, another class declaration may be one of the declarations of this class. This new class would then be a class attribute of the original class.

The first example of a class declaration is one in which the class contains no operational statements.

```
CLASS a;
  BEGIN
    REAL x;
    INTEGER y,z;
  END;
```

This class declaration links to a class unit "a" the variables "x", "y", and "z". If this class is to be used, it must be incarnated. This is accomplished by the statement NEW in the following way.

```
NEW a;
```

This statement will set aside an area in memory for a real variable "x" and two integer variables "y" and "z". This area may be referenced by a variable by using a reference statement.

```
r :- NEW a;
```

The variable "r" will as a result be pointing to the new incarnated area of class "a". The NEW statement thus creates an object which is an instance of class "a" and also starts executing any operational statements that appear inside class "a". This execution continues until the end of the class body or until a "detach" statement, which will be defined later, is encountered. Prior to the NEW statement, however, there must be a declaration defining "r" to be a

reference to class "a". The declaration statement,

```
REF(a) r;
```

will declare "r" as a reference to an incarnation of "a". It will reserve a storage location for "r" in memory of type reference. This area will be initialized as NONE meaning that no reference to an incarnation was established yet. The incarnation is then accomplished by "r :- NEW a;" as previously described.

At this point, values can now be assigned to the variables in this area. Suppose that a real value from input is to be placed in "x". This can be done by

```
r.x := INREAL;
```

This transfers a real value from input to the variable "x" in the incarnation of class "a", to which the variable "r" refers.

To summarize the discussion to this point, access to a class is available only via reference variables. A class instance is generated by a NEW statement. Each class instance has no name, only a reference. An array of class instances of class "a" can be accomplished by first declaring the array as follows.

```
REF(a) ARRAY arr(1:100);
```

This defines an array "arr" of one hundred elements, each of

which is a reference to a class instance "a". Now, one hundred class instances can be incarnated by a FOR statement.

```
FOR i:=1 TO 100 DO
  BEGIN
    arr(i) :- NEW a;
    arr(i).y := i;
  END;
```

Here, each incarnation of a class instance is formed with the "y" variable in each being set to the value "i". Now consider the following statement.

```
arr(1) :- r;
```

This will assign the reference of the incarnation of class "a" that was pointed to by "r" to the reference "arr(1)". In other words, both "arr(1)" and "r" reference the exact same incarnation or instance.

Consider the next class declaration.

```
CLASS box(length,width,height);
  INTEGER length,width,height;
BEGIN
  REAL PROCEDURE volume;
    volume := length * width * height;
  IF length<=0 OR width<=0 OR height<=0 THEN
    error;
END;
```

Also, suppose the following reference statement and incarnation was given.

```
REF(box) boxr;
boxr :- NEW box(3,4,5);
```

For the instance or incarnation of "boxr", the length, width and height are given the values three, four and five respectively through parameter passing. Also declared in this instance is a procedure name "volume". Finally, the operational statement, which checks to see that the length, width and height are greater than zero in this case, is executed. Every instance of this class will execute this operational statement. Each instance will have its own set of data (length, width and height). But, now turning back to the instance of "boxr", if the statement,

```
h :≡ boxr.height;
```

is executed, the previously declared integer variable "h" will be given the value of the attribute "height" of the reference "boxr", which in this case is the number five. Now, if the statement,

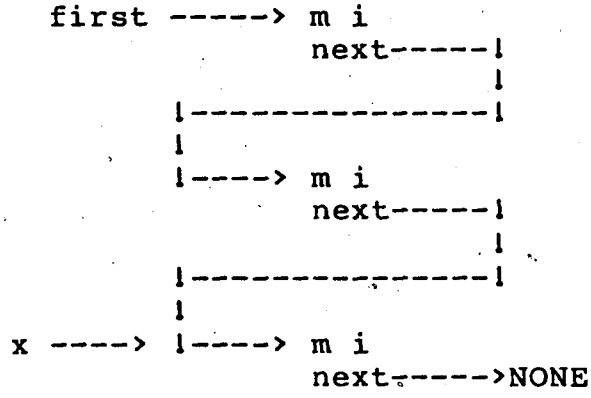
```
v := boxr.volume;
```

is executed, the following will occur. The procedure "volume" operates on the data of the instance "boxr". This means that the integer variable "v" will receive the value sixty.

A chain of instances of a class can be created by declaring a reference to the class inside the class itself. For example, consider the following piece of code.

```
BEGIN
  REAL ml;
  REF(c) x,y,first;
  CLASS c(m);
    REAL r;
  BEGIN
    INTEGER i;
    REF(c) next;
  END;
  ml := 1;
  first :- x :- NEW c(ml);
  x.next :- NEW c(ml);
  x :- x.next;
  x.next :- NEW c(ml);
  x :- x.next;
END;
```

Here, a class "c" can be referenced by "x" and "first" external to the class and by "next" internal to the class. Also shown is a multiple assignment "first :- x :- NEW c(ml);". The piece of code will generate a chain-like list as shown graphically below.



It is seen that the reference "first" always points to the

first instance of the class and "x" points to the last instance created. Also, the reference value "next" of the last instance will have the empty reference constant NONE. A loop can be used to run through the list by testing until an empty reference is found. The following code performs such a loop.

```
y :- first;
WHILE y =/= NONE DO
  BEGIN
    ...
    y :- y.next;
  END;
```

A "prefix" may be added to the front of a class declaration. This prefix is in fact another class name. The prefixed class is called a "subclass" of the prefix. This is shown below.

```
CLASS one;
  BEGIN
    INTEGER i1;
    i1 := 9;
    INNER;
    i1 := 10;
  END;
one CLASS two;
  BEGIN
    INTEGER i2;
    i1 := i2 := 20;
  END;
```

When class "two" is incarnated, every instance will get all the properties of class "one" plus the properties of class "two". When execution passes to class "two", the code before the INNER statement in class "one" is first executed.

The code of class "two" is then executed followed by the code after the INNER statement of class "one". A subclass is thus said to be "inner" to its prefixes. Therefore "two" is inner to "one". A heirarchy of subclasses can be formed by introducing a succession of declarations of subclasses. This can be shown as,

```
CLASS one ...;
one CLASS two ...;
two CLASS three ...;
one CLASS four ...;
```

By using subclasses, an organization of systems can be formed in different levels of abstraction. A subclass is equivalent to the class obtained by the concatenation of those classes that are on its prefix sequence. An object of a class resulting from a concatenation is a "compound object". The space occupied by the data structure of a compound object is the union of the spaces occupied by the data structures of the various classes in the prefix sequence. Suppose that the reference variable is declared,

```
REF(one) r;
```

The statement,

```
r :- NEW one;
```

is executed by placing a real copy of only "one" in storage with all the variables declared within class "one". The variable "r" then refers to this incarnation. However, if

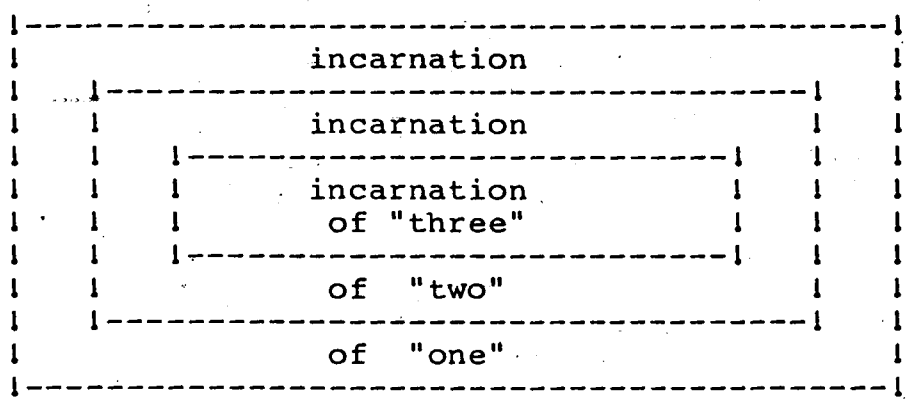
the statement,

```
r :- NEW two;
```

is executed, first an incarnation of class "one" is created. The subclass "two" is embedded in this incarnation. This is done by replacing the INNER statement of "one" by the initial operations of "two" followed by an INNER statement and the final operations of "two". This is thus the role of the INNER statement. In one case INNER is a dummy instruction when executed and in the other case it specifies where code may be inserted for a compound object. Likewise, the statement,

```
r :- NEW three;
```

will create an incarnation of class "three" embedded in class "two". Both of these classes are embedded in class "one". This is shown graphically below.



As indicated in the graph above, the subclasses are embedded in the incarnation of the comprehensive ones. This means

that each incarnation is a relatively self-contained unit in the general structure of the compound object created. Any variables of class "one" can be referenced by the variable reference "r" directly. For example,

r.i or r.b

However, variables in subclasses "two" and "three" must have additional "qualifications" to connect the name of the subclass to which the variable belongs. Two examples are shown below using the keyword QUA.

(r QUA two).m or (r QUA three).l

Here, variable "m" is from subclass "two" and variable "l" is from subclass "three". More on referencing variables will be covered in a later section.

Now, an example is presented which will define a stack. (See Ghezzi, 1982) This definition of a stack will allow the user to look at the top of the stack, insert an element on the stack, delete an element from the stack and check if the stack is empty. These operations are independent of the type of the elements that are stacked together. First, a class is used to describe the items that can be stacked.

```
CLASS stack_member;
  BEGIN
    REF(stack_member) next_member;
    next_member :- NONE
  END
```

All stackable objects will share an attribute, namely, a reference to the next item in the stack. The class "stack" describes only the operations applicable to all stackable objects.

```
CLASS stack;
  BEGIN
    REF(stack_member) first;
    REF(stack_member) PROCEDURE top;
      top :- first;
    PROCEDURE pop;
      IF NOT empty THEN
        first :- first.next_member;
    PROCEDURE push(e);
      REF(stack_member) e;
    BEGIN
      IF first=/=NONE THEN
        e.next_member :- first;
      first :- e;
    END;
  BOOLEAN PROCEDURE empty;
    empty := first==NONE;
  first :- NONE;
END;
```

The "stack" class contains a reference "first" to a stack member. The only statement in the class is the one that sets the reference "first" to point to NONE. Thus, when a class instance is incarnated, the result is an empty stack. The subprogram "top" is a function procedure that returns a reference. The reference is the stack member at the top of a nonempty stack. The subprogram "pop" is a proper procedure that simply deletes the top member of a nonempty stack. Procedure "empty" returns a "true" value if the stack is empty. Finally, procedure "push" will place a new stack member on the stack.

Stackable objects of a particular type can now be defined. For example, a stack of boxes can be defined as follows.

```
stack_member CLASS box(length,width,height);
  BEGIN
    ...
  END;
```

The class "box", as defined previously, is prefixed by "stack_member". Therefore, objects generated by

```
NEW box(...);
```

will have all the attributes of "stack_member", as well as the attributes of "box". "Box" is a subclass of "stack_member". Now, to create a stack of boxes, a variable reference of type "stack" must be declared by

```
s :- NEW stack;
```

This will create an empty stack. To push several boxes (b1,b2,b3) on the stack,

```
s.push(b1);
s.push(b2);
s.push(b3);
```

"s.top" will return the reference of the top element of the stack. "s.pop" will remove the top element from the stack.

A block in SIMULA 67 can also be prefixed by a class. Prefixing a block with a class name makes the attributes of the class visible to the block. For example, if a block

starts with

```
stack BEGIN
  ...
END;
```

the block has access to procedures "empty", "push", "pop", and "top". Thus, prefixing supports top-down modular design. The top-level class can contain only the global design decisions. Successive subclasses of this class can contain design decisions at lower levels of abstraction. At the lowest level, the program is prefixed by the detailed class. Thus, the different classes correspond to the different levels from which a given problem can be viewed. The prefixed classes of SIMULA 67 bring this mode of leveling design into a programming language. This feature can also be used as a powerful kernel language for the design of problem oriented languages.

The last topic to be covered in this section is the topic of virtual entities. A class has common attributes. But at times, an attribute may take a different meaning for different subclasses of the class considered. This can be accomplished by adding a virtual part to the declaration. The way this is done is by using the keyword "VIRTUAL:" followed by a proper or functional procedure. If, for example, an object of class "c" is given. Now suppose that an attribute "v" is specified as virtual in class "m" belonging to a prefix sequence of "c". An occurrence of "v"

is interpreted as virtual in "m" and in all classes inner to "m". The matching definition for the virtual attribute "v" is the innermost definition of "v". Below is an example program which shows the virtual concept. (See Ichbiah, 1972)

```
BEGIN
  CLASS real(real_part);
    REAL real_part;
    VIRTUAL: PROCEDURE show;
  BEGIN
    PROCEDURE show;
      BEGIN
        outfix(real_part,2,5);
        outimage;
      END
  END;
  real CLASS complex(image_part);
    REAL image_part;
  BEGIN
    PROCEDURE show;
      BEGIN
        outfix(real_part,2,5);
        IF SIGN(image_part)>0 THEN
          outfix('+i')
        ELSE
          outtext('-i');
        outfix(ABS(image_part),2,5);
      END
  END;
  REF(real) x;
  REF(complex) y;
  x :- NEW real(3);
  y :- NEW complex(2,4);
  x.show;
  y.show;
END
```

The procedure "show" is a virtual procedure in class "real". However, it is defined in both class "real" and class "complex". The definition in class "real" is used when "show" is called for "x". When "show" is called for "y",

two definition exist. The innermost definition, the one in class "complex", is the one that is used. Thus, the statement "x.show" will print "3.00" and the statement "y.show" will print "2.00+i4.00".

There are several points that should be made concerning the implementation of a dynamic object like a class. First, dynamic objects involve dynamic allocation which implies that some form of dynamic memory management must be provided by the compiler. This is no longer a simple stack allocation system since each object's existence is independent of all others. As a result, some form of memory recovery is needed. Another implementation point concerns pointer variables. These pointers can be implemented with almost no run-time checking. By having only one pointer to each data object, simultaneous multiple access is avoided without incurring the overhead of some synchronization mechanism. The only question at run-time is whether or not the pointer is empty.

F. Reference Variables

This section will deal with reference variables and remote identifiers in greater detail. First, the difference between "local" and "remote" access will be covered. Actions performed involving attributes of a particular object during the execution of the object deal with the local accessing of attributes. Accessing attributes of

other objects is termed remote accessing. However, since several instances of the same class may exist in memory at the same time, a given attribute is not uniquely specified by the name of its attribute identifier. Thus a remote access is accomplished by first selecting the particular object and then determining the attribute of that object. This is partly accomplished by the "reference" variable, which points to an object. A refernce to a newly created object is obtained by the execution of an object generator. This is done by using a NEW statement.

SIMULA 67 allows for the calculation of references and for their assignment to reference variables. This could bring about a reference to an object which belongs to a class other than the one intended. This is a very subtle error. In the majority of cases, except for this one, referencing errors are detected at compile time. That is why a qualification must be included in the declaration of each reference variable. This qualification indicates the class of an object to which the variable may belong. This qualification also is used to test for validity of reference assignments to that variable. Suppose first that the following piece of class code is given.

```
CLASS one;
...
one CLASS two;
...
one CLASS three;
...
```

Now suppose the following code is defined.

```
REF(one) a;
REF(two) b;
b :- NEW two;
a :- b;
```

The compiler will check to see if the previous code is valid. In the first assignment, "b" refers to the object that was intended. The next assignment, "a" is qualified by "one" and serves for remote access to attributes of that class. Also, each of these attributes are also present in an object of any subclass of "one", so "b" can be assigned to "a". If the statement,

```
b :- a;
```

were executed, the object referred to by "a" would have to be checked at run-time to see if it was actually a member of or subclass of "two". If it was, the assignment will not result is a run-time error. An assignment that would be rejected at compile time appears below.

```
REF(three) c;
c :- b;
```

Note here that "b" is qualified by "two" and "c" by "three". Trouble occurs since "two" and "three" are not in the same prefix sequence.

Remote accessing must be carried out without ambiguity and without loss of security. To do this, an object and an

attribute along with the qualification of the object must be known. This can be carried out in two ways. The first is by a "remote identifier" which consists of a reference variable and an attribute. For example, if "o" is an attribute of class "one" and "t" is an attribute of subclass "two" prefixed by "one", the remote identifiers "a.o", "b.o" and "b.t" are valid. However, "a.t" is not permitted since "t" is not in the scope of "one", the qualification of "a". Finally, a local qualification can be placed on a reference variable for accesses which ordinarily are not permitted.

```
a QUA two.t;
```

where "a" is given the local qualification of "two", is a legal reference.

The second way to accomplish remote accessing is by the "connection" mechanism. The format is,

```
INSPECT <object reference>
  WHEN <class identifier> DO <connection block>
  ...
  WHEN <class identifier> DO <connection block>
  OTHERWISE ...;
```

The qualification of the object is compared to the class identifiers in each WHEN clause until a class is found that belongs to the prefix sequence of the qualification. When found, the corresponding connection block is executed. This block contains the attributes of the object referred to by only mentioning their identifiers, exactly as with local

access. The connection mechanism therefore permits the same validity checks as for formal referencing.

In conclusion, 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. Design specifications of SIMULA 67 ensured that this check could be carried out entirely at compile time, thus avoiding the inefficiency of run-time checking. 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 a reference variable has the value NONE.

G. Quasi-Parallel Programs

The need may arise for different processes to act in parallel. Many simulations require parallel processes. However, since a single processor is used to carry out simulation, a quasi-parallel representation of a parallel program is used. This is accomplished by treating each process sequentially and using the system time to create the illusion of parallelism. The sequencing of events was accomplished in SIMULA by a list of event notices. The event notice contained, as previously explained, a reference to a process and a time when the process must be activated. The list was ordered in regards to time and the processes

were executed in order of event notice placement on the list. SIMULA 67 uses a class called SIMULATION to do the sequencing. This class will be explained in the next section. SIMULA 67 also provides the procedures "detach" and "resume" to accomplish quasi-parallel processing. This section will be devoted to this form of quasi-parallel processing.

The execution of a SIMULA 67 program involves the generation and execution of blocks, prefixed blocks, and objects. An object remains until the delimiter END is encountered or until a call to the procedure "detach" is issued inside the execution of the object. Once an END is encountered, the object is no longer active but still remains in memory. The object is said to be in a "terminated" state. If a "detach" is encountered, the object becomes "detached" from the component to which it belonged. This process then becomes an independent component. Thus, the main program and all detached objects are components, and these components are what form a "quasi-parallel program".

Unlike a subroutine, which is subordinate to its caller, a "coroutine" is a module coordinate with other modules. 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 the reference variables. A producing coroutine may

assign values to any variables and the consuming coroutine can access them. Coroutines may also change any global variables. When an object is generated, it has a procedure-like relationship to the block which generated it. Control automatically returns to the generator upon passage through the end of the object. The object does not necessarily know the identity of its generating block. As a result, a resume instruction can not be used to achieve the effect of a coroutine exit. The statement,

```
DETACH;
```

is provided by which a generated object can return to the generator. When a new instance of this class is created by the NEW statement, the sequence of operational statements will be executed as far as the statement DETACH. The DETACH statement causes this incarnation to be interrupted and execution to continue at the part of the program which caused the class to be incarnated. As a consequence of DETACH, the unit behaves as a coroutine that can be resumed and, in turn, can resume other coroutines. A detached class can be resumed by the statement,

```
RESUME(r);
```

where "r" is a variable that refers to a detached class. Now, the object is again in a subroutine position with respect to the caller and has an obligation to return to

either by a DETACH instruction or by going through its own END.

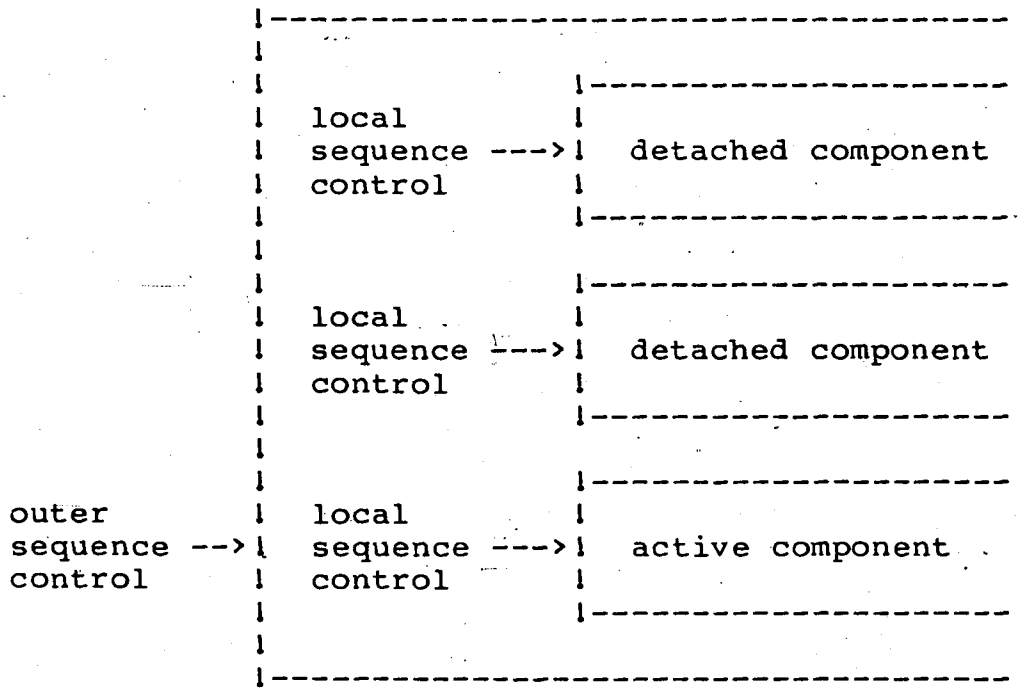
A main program, therefore, can establish a coroutine relationship with an object that it has generated using the RESUME/DETACH mechanism instead of the more symmetric RESUME/RESUME mechanism. In the RESUME/RESUME mechanism, control of one generated object is passed by using a RESUME instruction to another generated object. In turn, that generated object resumes another generated object and so on. The generated objects do not return control back to the generator since a DETACH statement is not present. On the other hand, in the RESUME/DETACH mechanism, the generated object remains subordinate to the generator, or the main program. As a result, this mechanism is sometimes referred to as a "semi-coroutine". But, this can still act as a full coroutine with respect to a group of other generated objects with which it communicated by means of RESUME statements. If any of the group issues a DETACH, control returns to the master program which originally called the particular member of the group. Thus, the coroutine issuing the RESUME imposes on the resumed coroutine its own responsibility to eventually pass control back to the original caller by means of a DETACH.

In summary, a typical coroutine object goes through the following history. First, the object, upon generation, performs the operations of its class body. The object

eventually executes a DETACH which returns control to the point at which the object was generated. The object is detached, but not terminated. The object becomes suspended with its state preserved. Control then returns to the object on a RESUME, to the point where the object left off. The object may again relinquish control, either by a DETACH or by a RESUME, and become detached again. Finally, the object terminates by executing its END statement. This has the same effect as a DETACH, except that the object may not be reactivated by a RESUME. However, if the RESUME/RESUME mechanism is used without using a DETACH, the object must not terminate itself directly since all other objects are suspended. By this, no other objects will be able to be reactivated. An object, once it is terminated, remains in existence as an item of data, which may be referenced by remote identification of its attributes, including procedure and function attributes.

At any one time only one component in the program is being executed. All other components are temporarily suspended. A quasi-parallel program has two types of control that effect sequencing when a DETACH or RESUME is encountered. The "outer sequence control" points to the statement that is being executed at the given time. The "local sequence control" of a component of the program points to the actual executing statement if the component is active, or points at the statement which execution will be

resumed if the component is detached. A graphical example appears below.



Here it is seen that the local sequence control of the active component is the exact same as the outer sequence control of the quasi-parallel program.

Using these two controls, the effects of the DETACH and RESUME will be covered. When an object calls DETACH for the first time after being initially attached, the object becomes an independent component. The execution of the object is suspended and its local sequence control is placed after the DETACH statement. On the other hand, the outer sequence control is placed after the instruction that caused the generation of the object. The RESUME(c) statement will cause the execution of the object referenced by "c" to be

resumed. The component containing the RESUME will have its local sequence control placed after the RESUME statement. The outer sequence control is placed at the local sequence control of the referenced object "c". Finally, a terminated object cannot be resumed since it has no local sequence control. Therefore, the reference of a RESUME statement must refer to a detached object.

In SIMULA 67, coroutines are executed immediately as they are created. When the instance is created, its activation record is placed on a stack. A problem occurs in the RESUME/RESUME mechanism where the communication between coroutines is symmetric. Since this type of coroutine is symmetric and the management of a stack is asymmetric, coroutine activation records cannot share a stack. Instead, each coroutine activation record occupies the bottom of its own stack, which grows and shrinks as the coroutine enters BEGIN/END blocks, activates other coroutines, or terminates. Thus, the execution-time representation of this type of coroutine requires multiple stacks.

The RESUME/DETACH coroutine mechanism uses semi-symmetric communication between coroutines. Here, a set of coroutines exists, but the coroutines do not pass control between each other as in the RESUME/RESUME mechanism. Instead, these all pass control between themselves and the main program, which acts as a controller program. A popular way to implement this type of coroutine is by using a

"spaghetti" stack. This stack contains activation records needed for coroutine reactivation. These records include pointers to the coroutine's creator, its activator, its reactivation point, a pointer to the start of the activation record and other temporary values. The number of references to an activation are kept during the execution. When this number becomes zero, the activation record is removed. Usually, a coroutine only contains one pointer to its body which indicates where it will resume execution when activated. This pointer is termed a "moving" activation point. A pointer to its body for each coroutine which activates this coroutine can be saved. These pointers are termed "static" activation pointers. The coroutine may resume at different places depending upon which coroutine reactivates it. When a typical procedure ends, its activation record must be removed from the stack. But, when a coroutine ends, the activation record must not be removed. By keeping this record on the stack, another coroutine or procedure may be blocked if there is not enough room for the stack to grow. As a result, the part of the activation record which is used for coroutine information is copied to another part of the stack where there is enough space. This copying leaves gaps in the stack. Because of these gaps, the name "spaghetti" stack came about.

An example of a quasi-parallel program using coroutines is now presented. (See Ghezzi, 1982) The program is a card

game in which four players will use the same strategy.

```
BEGIN
  BOOLEAN gameover;
  INTEGER winner;
  CLASS player(n,hand);
    INTEGER n;
    INTEGER ARRAY hand(1:13);
  BEGIN
    REF(player) next;
    DETACH;
    WHILE NOT gameover DO
      BEGIN
        create a move;
        IF gameover THEN
          winner := n;
        ELSE
          RESUME(next);
      END
  END;
  REF(player) ARRAY p(1:4);
  INTEGER i;
  INTEGER ARRAY cards(1:13);
  FOR i:=1 STEP 1 UNTIL 4 DO
    BEGIN
      generate cards for player i in array card;
      p(i) :- NEW player(i,cards);
    END;
  FOR i:=1 STEP 1 UNTIL 3 DO
    p(i).next :- p(i+1);
  p(4).next :- p(1);
  RESUME p(1);
  print winner's name;
END
```

Four players are created by the first FOR loop, "p(i) :- NEW player(i,cards)". Each player is detached on initial generation. These players are then linked together by the next FOR loop. This link is closed by linking the fourth player back to the first player. Finally, player number one is resumed from the point he was detached. The four players will function in parallel of each other but their statements will actually be carried out piece by piece in sequence.

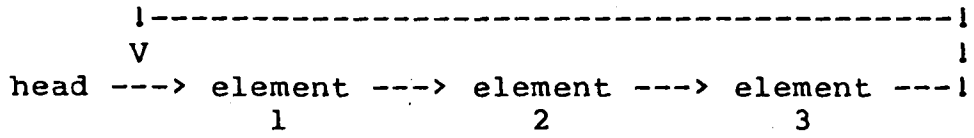
Therefore, four coroutines, the players, exist. These coroutines are incarnations of the same class, however, in general, coroutines may be incarnations of different classes. The first player will start the game by performing some move. The procedure to perform a move is not shown. The player will then check the "gameover" variable. If it is "false", the player will resume the next player in the linked list. The next player will then continue the game and so on. All four players will continue to play with no control from the outside. As soon as a player wins the game, variable "gameover" is set to "true". The execution of the coroutine instance will then terminate. Control will return back to the main program at the instruction after the resumption of the first player. As a result, the main program will then print out the name of the winner and halt.

In the game playing program it was seen that a closed linked list was created. This was done by linking together various class incarnations by reference variables. Each player was an individual incarnation. Each player was an element of the list. Linked lists come in various types. Each linked list usually has a "head" pointer which points to the first element of the list. A typical "singularly linked list" would appear as,

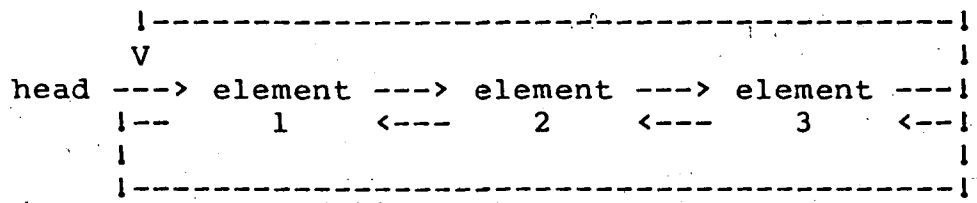
```
head ---> element ---> element ---> element ---> NONE
             1              2              3
```

Here, there are three elements linked together. Each

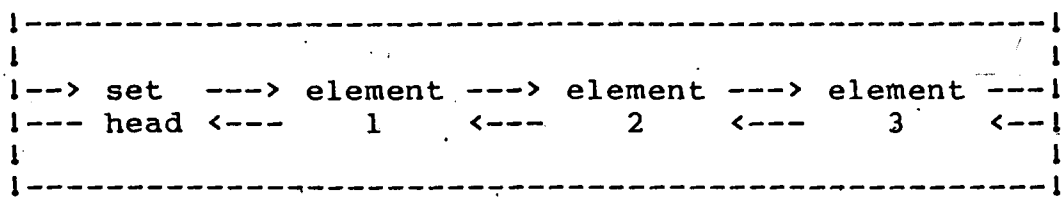
element, therefore, has contained in it a reference to another element. "Head" points to the first element and the last element points to NONE. If the last element would have pointed back to the first element,



a "singularly circular linked list" would have been created. Another form of a linked list is a "doubly circular linked list". Here each element contains a reference to the succeeding element and also to the preceeding element.



Here, "head" is only a reference to the first element. However, it may be necessary in some instances to create a dummy element called the "set head". This set head would actually be an incarnation of a class and would be at the head of the list.



SIMULA 67 provides a mechanism, in the form of a class, to handle list processing. The class is named "simset" and contains the following.

```
CLASS simset;
BEGIN
  CLASS linkage;
  BEGIN
    REF(linkage) succ,pred;
    REF(link) PROCEDURE suc; ...
    REF(link) PROCEDURE pre; ...
  END;
  linkage CLASS link;
  BEGIN
    PROCEDURE out; ...
    PROCEDURE follow(x);
      REF(linkage) x; ...
    PROCEDURE precede(x);
      REF(linkage) x; ...
    PROCEDURE into(s);
      REF(head) s; ...
  END;
  linkage CLASS head;
  BEGIN
    REF(link) PROCEDURE first; ...
    REF(link) PROCEDURE last; ...
    BOOLEAN PROCEDURE empty; ...
    INTEGER PROCEDURE clear; ...
    INTEGER PROCEDURE cardinal; ...
    succ :- pred :- THIS head;
  END;
END;
```

By using "simset", a doubly linked list will be created with a "set head" element. When the list is created but still empty, the only element of the list will be the "set head" which will have both succeeding and preceding references pointing to itself. A created element that is not in a list will have its succeeding and preceding references pointing to NONE.

The class "simset" contains three classes as attributes. The class "linkage" contains two reference variables "suc" and "pre". The two procedures that are defined are used to get the references for the two reference variables. Details of these and all other procedures used in class "simset" are not presented but rather a brief explanation of what each does is included. The subclass "head" creates an incarnation which is the head of the linked list. This is done by the statement "succ :- pred :- THIS head". The expression "THIS head" is a reference to the "head" object being executed. Procedure "first" will establish a reference to the first element of the list. Procedure "last" will establish a reference to the last element of the list. If no elements are present, the "set head" references will be NONE. Procedure "empty" will be set to "true" if there are no elements in the linked list. Procedure "clear" will delete all the elements of the list. Finally, procedure "cardinal" will return the total number of elements in the linked list. The subclass "link" contains procedures to view an element of the list. All the attributes of this class are procedures. These attributes are accessible by means of remote identifiers. The procedure "into(s)" places an incarnation of a class at the end of the list "s". The procedure "follow(x)" will place an incarnation behind the list element "x". The procedure "preced(x)" will place an incarnation in front of the list

element "x". Finally, procedure "out" removes an incarnation from the linked list.

Now that the class "simset" has been defined, it can be used to manipulate lists in the following way. To have the capabilities of class "simset" inside a block, the name "simset" must prefix the block.

```
simset BEGIN
  ...
END;
```

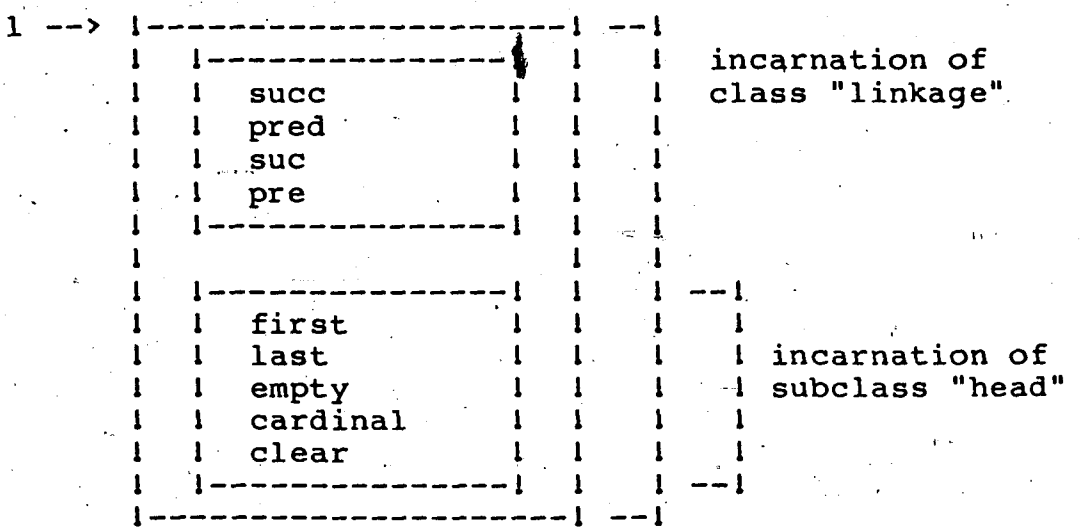
If a variable "l" refers to a "head" by,

```
REF(head) l;
```

a new head of a list can be created by,

```
l :- NEW head;
```

This will create an instance that contains both class "linkage" and class "head".



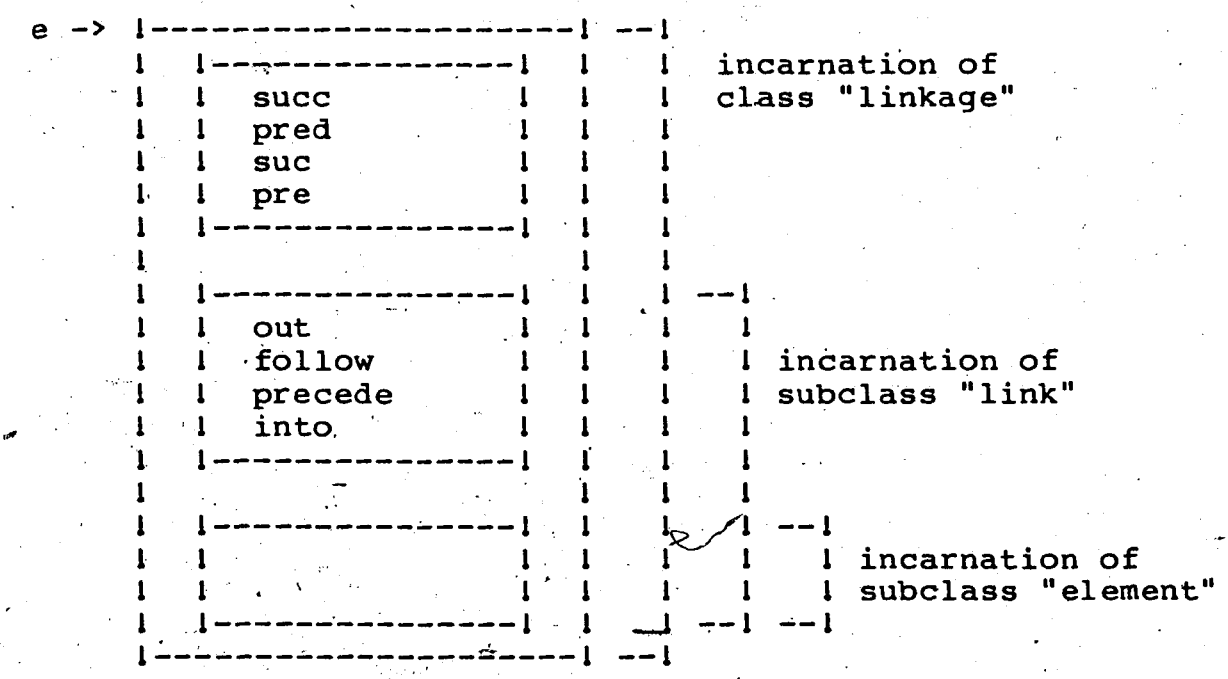
Any of the procedures of these classes can be used by referencing through variable "l". An example of this is "l.first" or "l.pred". Now the linked list is created. To add elements to this linked list, an element must be defined. The elements of the list must be a subclass of class "link". This is shown below.

```
link CLASS element;
BEGIN
  ...
END;
```

This class represents an element that may be placed in the list. An element may be of any type. Class "simset" works independent of the types of elements. The following statements will create a reference to a "link" and create an incarnation of an element.

```
REF(link) e;
e :- NEW element;
```

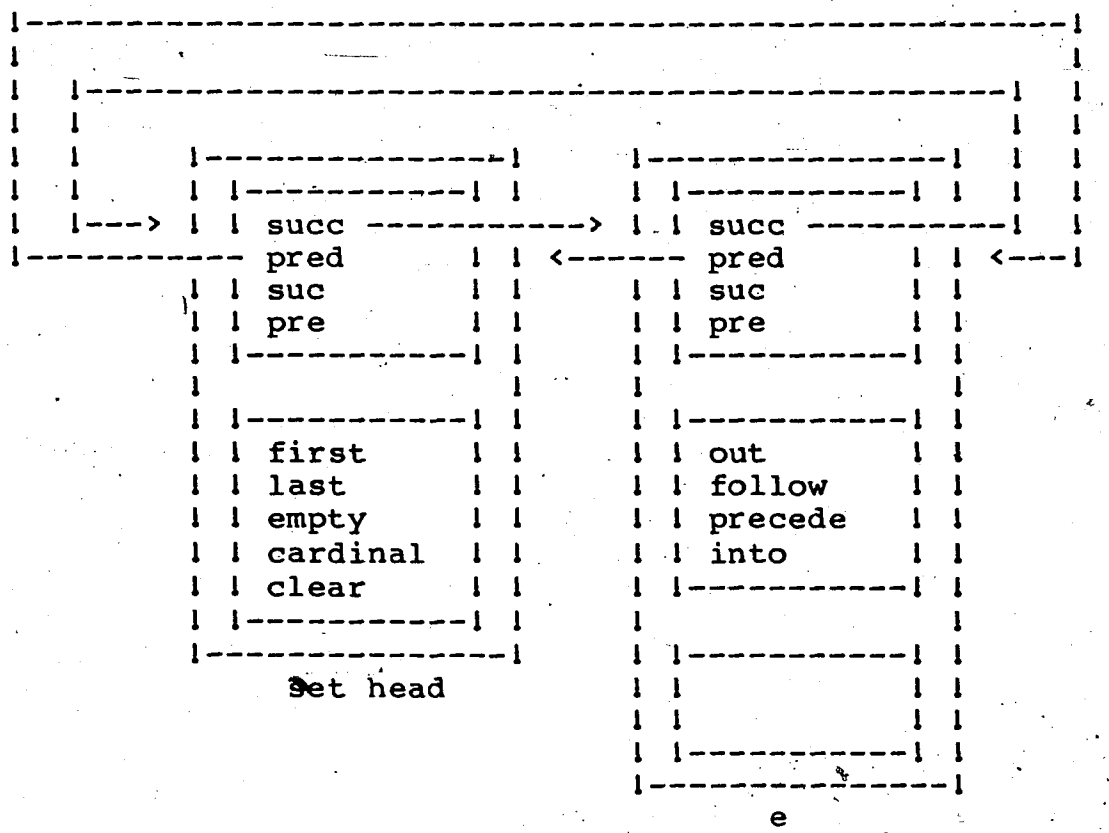
Graphically, this creates an object that contains both class "linkage" and "link" along with class "element".



Again, any procedures or variables in either "linkage" or "link" can be referenced by using the reference "e". An example of this is "e.into". However, to reach variables or procedures in the "element" class, the qualified "e QUA element" must be used. Suppose now, that the incarnation pointed to by "e" is to be put in the list "l". This is accomplished by the statement,

```
e.into(l);
```

Graphically this is shown.



Thus, in general, each element of the list gives rise to incarnations of the comprehensive class "linkage" and subclass "link" along with their variables and procedures.

The section on list processing is completed with a program to sort words given in the input. A linked list will be used to store the words as they come in.

```
BEGIN
simset BEGIN
  REF(head) l;
  REF(link) nextword,current;
  TEXT inputword;
  link CLASS word(t);
    VALUE t;
    TEXT t;
  BEGIN
  END;
  l :- NEW head;
  INIMAGE;
  inputword :- SYSIN.IMAGE;
  nextword :- NEW word(inputword);
  nextword.into(l);
  INIMAGE;
  WHILE NOT ENDFILE DO
    BEGIN
      current :- l.first;
      inputword :- SYSIN.IMAGE;
      WHILE current =/= NONE DO
        BEGIN
          IF inputword<(current QUA word).t THEN
            BEGIN
              nextword :- NEW word(inputword);
              IF nextword =/= NONE THEN
                nextword.precede(current)
              ELSE
                nextword.into(l);
              INIMAGE;
            END
          ELSE
            current :- current.suc;
        END
      WHILE NOT l.empty DO
        BEGIN
          current :- l.first;
          OUTTEXT((current QUA word).t);
          OUTIMAGE;
          current.out;
        END;
    END;
END;
```

The program contains one big "simset" block. Inside this block, a list referenced by "l" and two link references "nextword" and "current" are declared. The elements are words. The prefixed class "word" is an empty class as shown

by nothing being in the BEGIN/END delimiters. Only a parameter "t", which is a word of text, is inside the class "word". The program then creates a set head incarnation,

```
l :- NEW head;
```

followed by the incarnation of the first element,

```
nextword :- NEW word(inputword);
```

This word is then placed into the list by the statement

```
nextword.into(l);
```

The WHILE loop will continue to read words from the input and create corresponding incarnations for each word. This incarnation of an element will be placed into the list in alphabetical order. Finally, the last WHILE loop will go through the list and print out each word. The incarnation will be deleted after the word is printed.

H. Simulation

All models dealing with simulation can be described by the quasi-parallel coroutine mechanism, along with the list class "simset". However, SIMULA 67 provides the class "simulation" in order to simplify the running of simulation models. In a simulation model, multiple processes interact at any given instant of time. These processes must be scheduled in order to activate at a certain time. Each

process is tagged by the scheduler with an event notice which will contain the name of the process and the time the process is to activate. The event notice of all processes are then put into a list which is sorted in chronological order with the process to be activated next at the head of the list. The class "simulation" therefore is declared as a subclass of class "simset". This allows the "simulation" class to use the list processing capabilities of class "simset" for queuing event notices and processes. To achieve quasi-parallelism, the scheduling mechanism uses the "detach" and "resume" procedures.

The contents of class "simulation" will be discussed by first showing a rough, outline form of the class. (See Ichbiah, 1972)

```
simset CLASS simulation;
BEGIN
  link CLASS event_notice(evtime,proc);
    REAL evtime;
    REF(process) proc;
  BEGIN
    ...
  END;
  link CLASS process;
  BEGIN
    REF(event_notice) event;
    DETACH;
    INNER;
    PASSIVATE;
    error;
  END;
  REF(head) SQS;
  procedure hold ...;
  procedure passivate ...;
  procedure activate ...;
  process CLASS main;
  BEGIN
    1: detach;
```

```
        GOTO 1;
    END;
    REF(main) m;
    SQS :- NEW head;
    m :- NEW main;
    m.event :- NEW event_notice(Ø,m);
    m.event.into(SQS);
END
```

Inside class "simulation" is defined a class "event_notice" which is a subclass of class "link". These event notices, therefore, are allowed to be placed on a list. The list is called the "sequencing set" of the simulation model. The parameter "proc" represents the process that this event notice refers to. The parameter "evtime" represents the time the corresponding process is to be activated. The sequencing set is sorted by increasing value of variable "evtime". Thus, the sequencing set represents the simulation time axis. The head of the sequencing set is referenced by the variable "SQS". Also in class "simulation" is the class "process" which again is a subclass of class "link". This allows processes to be put in lists. The reference variable "event" will point to the corresponding event notice of the process. The rest of class "simulation" defines the sequencing set head "SQS" and creates the sequencing set.

If simulation is needed, the block of the program is prefixed by class "simulation".

```
simulation BEGIN
  ...
END;
```

Once this is done, processes may be activated or interrupted. Thus, there are no longer coroutines being detached or resumed. A process is defined in a declaration of a class as follows.

```
process CLASS name(formal);
  BEGIN
    ...
  END;
```

A reference variable to the process can be declared by

```
REF(name) p;
```

Finally, a process "p" can be incarnated by

```
p :- NEW name(parameters);
```

To reiterate, a process of a simulation run is declared by prefixing the class name by the class "process". This newly declared process can now be referenced as any other class by using REF and incarnated by NEW.

Now that the declarations of processes are included in the simulation block, the simulation will be accomplished by the main program, or block, generating incarnations of processes and interacting with them. To provide for this interaction, it is necessary to be able to schedule the main program. Since the main program is not a process, the process "main", which appears in class "simulation", as shown, is used to accomplish the interaction. A simulation then progresses as follows. It was seen before that the

initial operations of class "simulation" will create the sequencing set and initialize it to contain the event notice for the process "main". Thus, when the main program is being executed, the process "main" will correspond to the first event notice in the sequencing set. The main program will generate and schedule other processes by using the scheduling procedures. These procedures will be covered later. Notice that the first instruction of class "process" is a DETACH. This sends control immediately back to the main program after the main program generates a process. The main program can pass control back to the created process by performing a scheduling procedure such as ACTIVATE. Control will then be passed back to the process next scheduled for execution. Simulation will continue with the passing of control between the main program and processes until the end of the block is found. At that point, the simulation will end.

In SIMULA 67, a process can be in one of four different states. The first event of a sequence set points to the process that is currently executing. This process is "active". Only one process can be active at any moment of time. The other processes whose event notices of the sequence set are scheduled to be executed at some later time are said to be "suspended". A process that has no event notice in the sequencing set is "passive" unless it has reached the end of its execution. In that case, the process

has executed the END statement of its class process body and is said to be "terminated".

When a process is incarnated by a NEW statement, the state of the process becomes passive. A process, for example "p", can be switched from a passive state to an active state by the statement,

```
ACTIVATE p;
```

This process "p" will become active immediately, at the present simulation time. A process also can be activated at a later time by the statement,

```
ACTIVATE p AT t;
```

where "t" is a real value that is greater than the current simulation time variable "TIME". Process "p", as a result, will be entered in the sequencing set with a value "t" as starting time "evtime". The process will now be in a suspended state until time "t" arrives. At that time, process "p" will automatically be started. Another way to accomplish this is by,

```
ACTIVATE p DELAY dt;
```

where "dt" is added to the current TIME to get the starting time of process "p". In either case, suppose that there are other processes with the same activation time. The process "p" would be placed after the other processes with the same

activation time on the sequence set. Therefore, process "p" would activate only after all the other processes with the same activation time were finished. If process "p" was to be activated before all the other processe with the same activation time, the statement,

```
ACTIVATE p AT t PRIOR;
```

or,

```
ACTIVATE p DELAY dt PRIOR;
```

will place process "p" on the sequence set before all the processes with the same activation time. To place a process "p" somewhere in between the processes with the same activation time, the statement,

```
ACTIVATE p BEFORE p1;
```

or,

```
ACTIVATE p AFTER p1;
```

can be used. These statements place process "p" on the sequence set either before or after the process "p1". If the process is to be started right after the current process is inactivated, the statement

```
ACTIVATE p AFTER CURRENT;
```

is used. The procedure CURRENT returns a reference to the active process.

All the above ACTIVATE statements work only if the process "p" is passive. If "p" is suspended or active, the ACTIVATE statements are ignored. IF "p" is terminated, the

program will abort. To set up an event notice for an active or suspended process, any one of the following statements can be used.

```
REACTIVATE p;
REACTIVATE p AT t [PRIOR];
REACTIVATE p DELAY dt [PRIOR];
REACTIVATE p BEFORE p1;
```

or,

```
REACTIVATE p AFTER p1;
```

Because, at any istant of time, only one process can be active, an active process can only be interrupted by itself. Thus, no inactive process, not even the main program, can activate an inactive process. An active process becomes passive by the following statement,

```
PASSIVATE;
```

The process can become active again if another process executes an ACTIVATE statement for that process. For example, if process "p" becomes passive by executing a PASSIVATE statement, it can be activated by an "ACTIVATE p" statement in the current active process. Another statement,

```
HOLD(dt);
```

changes the active process into a suspended process. The process will be reactivated automatically at "dt + TIME". The last way to interrupt an active process is to terminate it. The process is terminated when the END statement of the class process incarnation is executed. Once terminated, a

process can never be reactivated. Any attempt to reactivate it will result in a program abort. Even though a process is terminated, all process attributes can be accessed though remote variables.

A process "p" which is suspended can be made passive by the statement,

```
CANCEL(p);
```

This, as a result, removes the event notice of process "p" from the sequencing set.

A simulation program is now presented. The card game program, as presented previously using coroutines, is now presented using the simulation mechanism. Notice that only a few changes were made. First, the word "simulation" now prefixes the entire block. The class "player" is now a process. The DETACH statement inside the process is no longer needed. Finally, both RESUME statments have been changed to ACTIVATE statements.

```
simulation BEGIN
  BOOLEAN gameover;
  INTEGER winner;
  process CLASS player(n,hand);
    INTEGER n;
    INTEGER ARRAY hand(1:13);
  BEGIN
    REF(player) next;
    WHILE NOT gameover DO
      BEGIN
        create a move;
        IF gameover THEN
          winner := n;
        ELSE
          BEGIN
```

```
                ACTIVATE next AFTER CURRENT;
                PASSIVATE;
              END;
          END
      END;
      REF(player) ARRAY p(1:4);
      INTEGER i;
      INTEGER ARRAY cards(1:13);
      FOR i:=1 STEP 1 UNTIL 4 DO
        BEGIN
          generate cards for player i in array card;
          p(i) :- NEW player(i,cards);
        END;
      FOR i:=1 STEP 1 UNTIL 3 DO
        p(i).next :- p(i+1);
      p(4).next :- p(1);
      ACTIVATE p(1);
      print winner's name;
    END
```

I. Files

Up until this point, all examples dealing with input or output dealt with standard input/output. However, there are times when information must be placed in a file or must be read from a file. SIMULA 67 coordinates each file to an incarnation of a class. The procedures belonging to the class enable records to be transmitted from the program to the file or vice versa. The class is called "basicio". This comprehensive class contains many procedures already described, such as ININT, INTEXT, OUTFIX, SETPOS, POS, and INIMAGE, just to name a few.

By the declaration,

```
REF(outfile) f;
```

the following statement,

```
f :- NEW outfile("myfile");
```

can be executed. This creates an incarnation of an "output" class. The parameter of this output class is the actual file, called "myfile" in this case, that will be created. Every file that is created must first be opened. This is accomplished by

```
f.OPEN(BLANKS(80));
```

which opens the output file and also creates an output buffer of eighty characters. To place values in the buffer, the same commands as used for standard output are used. However, the commands are prefixed by the reference to the specified outfile. The following shows an example of this.

```
f.OUTFIX(r,a,w);
```

or

```
f.OUTTEXT('This is a text');
```

To place the contents of the buffer into the actual buffer, the statement,

```
f.OUTIMAGE;
```

is used. Finally, the file must be closed when finished. The statement,

```
f.CLOSE;
```

will close the file referenced by "f".

An input file is manipulated in exactly the same way except that the reference is to an "infile" instead of an "outfile".

The above definitions work on sequential files. Two restrictions on sequential files are present. The first is that a file must either be for reading, an "infile", or for writing, an "outfile". At any instant, only one record can be manipulated.

SIMULA 67 can also handle direct access files stored on disk. Here, records can be read from or written directly to a specified location. However, all records are restricted to being the same length. Each record is numbered and can be addressed. A direct access file, called "mydirect" in this example, is referenced and incarnated by the following statements.

```
REF(directfile) d;
d :- NEW directfile("mydirect");
```

As with a sequential file, a direct file must also be opened before it is used and closed when finished. The file is manipulated by using the addresses of wanted records. For example,

```
d.LOCATE(n);
```

will address record at address "n". This inturn will set the value of the record counter, "d.LOCATION", to be "n". As a result, the statement,

```
d.INIMAGE;
```

will transfer that record into the buffer, "d.IMAGE". On the other hand, the statement,

```
d.OUTIMAGE
```

will transfer the buffer contents to the file at address "n".

In general, access to records is accomplished by only using one key, the numerical address of the location of the record that is wanted.

BIBLIOGRAPHY

Dahl, O-J. "Discrete Event Simulation Languages". Programming Languages, F. Genuys, Academic Press, 1968 : 349-395.

Dahl, O-J., Dijkstra, E. W., and Hoare, C. A. R. Structured Programming, Academic Press, 1972.

Dahl, O-J. and Nygaard, K. "SIMULA-An ALGOL-Based Simulation Language", Communications of the ACM, Vol. 9, 9, September 1966 : 671-678.

Ghezzi, C. and Jazayeri, M. Programming Language Concepts, New York: John Wiley, 1982.

Horowitz, E. Fundamentals of Programming Languages, Computer Science Press, 1983.

Ichbiah, J. D. and Morse, S. P. "General Concepts of the SIMULA 67 Programming Language", Annual Review in Automatic Programming, Vol. 7, 1972 : 65-93.

Lamprecht, G. Introduction to SIMULA 67, Freidr. Vieweg & Sohn, 1983.

McNeley, J. "Simulation Languages", Simulation, Vol. 9, 2, August 1967 : 93-98.

Nygaard, K. and Dahl, O-J. "The development of the SIMULA Language", History of Programming Languages, R. Wexelblat, Academic Press, 1981 : 439-493.

Palme, J. "Experience from the Standardization of the SIMULA Programming Language", Software-Practice And Experience, Vol. 6, 1976 : 405-409.

Palme, J."Uses of the SIMULA Process Concept", Software-Practice And Experience, Vol. 12, 1982 : 153-161.

Wegner, P. Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, 1968.

VITA

Andrew Joseph Tanhauser was born in Bethlehem, Pennsylvania on December 13, 1958 of the parents Joseph and Caroline Tanhauser.

Mr. Tanhauser received a Bachelor of Science degree with a double major in Computer Science and Mathematics in 1980 from Moravian College. He began his graduate studies on a part-time basis while remaining fully employed. However, he finished his degree on a full-time basis in his final semester.

Mr. Tanhauser was employed at the Bethlehem Steel Corporation as a research technician from 1980 to 1983. After that, he was employed at National Systems Analysts as a system design engineer in 1983. He is presently teaching computer science courses at Moravian College as a part-time instructor. He also taught computer science courses at the Northampton County Area Community College.