Business Applications

D. TEICHROEW, Editor

A Language for Modeling and Simulating Dynamic Systems

R. J. PARENTE AND H. S. KRASNOW
*International Business Machines Corporation,**
Yorktown Heights, N.Y.

The general objective of this language is to facilitate both the modeling and experimental aspects of simulation studies. The ability to represent systems containing highly interactive processes is an essential feature. The nature of the language, and the role of the process concept, is presented by means of an extended example.

1. Introduction

Development in simulation languages has taken place for almost a decade, with interest increasing rapidly in recent years. Well over a dozen general purpose languages have been developed, and several of these are currently operational.

The advent of a new generation of computers, coupled with continued growth of interest in simulation as a tool for decision making, has helped to motivate the experimental language discussed in this paper. This language is indebted to prior languages which first clarified and embodied the basic concepts of discrete system modeling. But this language is an attempt to extend these concepts so as to provide, within a single system, both freedom for modeling and experimentation and a built-in concept of modularity.

In this paper the general orientation of the language is presented, rather than a formal definition. Emphasis is placed upon those features which contribute most directly to the overall structure of the language, with limited attention to the many details which must also be provided. Most of the discussion is based upon an example of a simulation study.

This simulation language is an experimental development of the Advanced Systems Development Division of IBM. A prototype processor is currently being implemented.

* Advanced Systems Development Division

2. Elements of a Simulation Study

System studies are frequently conducted with a representation, or model, of the system. Experiments conducted with such a model allow a systems analyst to observe performance under specified conditions. The main steps in conducting a simulation study are:

1. Define the problem.
2. Design an experiment.
3. Construct a model.
4. Specify the conditions of the experiment.
5. Observe the performance of the model.
6. Analyze the observations.

It is presumed that a substantive problem, or set of problems, exists, and that simulation is the tool appropriate to the study of the problem(s). The presumption is nontrivial. A simulation language itself is of no direct assistance in determining when to simulate or to what detail a model should be constructed. This determination rests upon the skill of the analyst, as does the design of the experiment or the families of experiments by which the problem is to be analyzed.

It is not feasible to define a simulation experiment fully without reference to a model or to the facilities of the computing system on which the model is to be run. Hence, a simulation language begins to influence the study in its earliest stages. From then on, it is an inseparable partner in the analysis. Its effects range from subtly influencing the model structure to directly imposing limitations on what can and cannot be done with the model.

The list of steps suggests a strict linear ordering to the stages of a simulation study, but in practice there is a need for extensive iteration over various stages. For example, an experiment cannot be fully defined until the model has been constructed; yet the *full description* of the system to be simulated requires information concerning the design of the experiments to be conducted with the model. At another stage, a complete experiment may often consist of multiple runs of a model, each run requiring reinitialization and start-up, as well as separate observations and analyses. Also, an outer loop exists wherein the analysis of experimental results leads to the redesign of experiments or to the design of new experiments with concomitant modifications of the model. A simulation language should contribute at all of these stages of the study and, in particular, should facilitate the frequent modification and change that is characteristic of the use of simulation for studying operational problems.

3. General Approach of the Language

This language is intended to serve the user in all stages of a simulation study. Its capabilities may be classified in two major areas: (1) The descriptive capabilities are designed to facilitate the clear and direct representation of a dynamic system. These may be referred to as the modeling capabilities. (2) The experimental capabilities serve for the expression and execution of a simulation experiment.

3.1. MODELING. There are two aspects to the representation, or modeling, of a system: The first is concerned with the identification and specification of each type of component that will exist in the model; the second pertains to the specification of the model dynamics.

Any system component that is to be manipulated or observed may be considered an entity. Some simple examples of entities in a factory are a machine, an order, and a unit of material. Each type of entity is described, i.e., its structure is specified, by associating with it certain characteristics, or attributes, that are of interest for a particular system representation. The values of the attributes determine the state of that entity, and the state of all the entities determines the state of the model.

Each entity of a type identified as SET may be utilized to arbitrarily group entities. The value of an attribute LIST_COUNT, of each SET is the number of entities that are grouped in that SET. All or selected members of a SET may be referenced.

The description of the state of a system at an initial simulated time, T_0, or at any time, T_i, is not complete without a description of the states of the processes that are taking place. A process is a form of entity that is not fully specified by its associated attributes; it possesses dynamic characteristics; i.e., a behavior pattern is associated with it. The user specifies this behavior pattern, using the statements of the language, for each type of process that may enter the model. Like other entities, however, a process enters the model (when it is started), it exists for a period of time (while it is taking place), and then it ceases to exist (when it stops).

Entities of each of the specified types are entered into the model, or "created." They exist in the model for a period of time and then leave it, or are "destroyed." When a SET is created, it is "empty"; i.e., there are no entities in it and the value of its attribute, LIST_COUNT, is zero. A new process is created each time a process of a given type is started. Entities, including sets and processes, may be created by an initializing process or at any time during the running of the model. Entities may be put into a SET any time after that SET has been created.

The identification and specification of each type of entity that is to be represented in the model will be referred to as the "component description" section of the model. In this section, each type of process is identified but is only partially specified; attributes are associated with it, but the actions that represent its behavior pattern are specified, with the behavior patterns for all other process types, in a "behavior description" section of the model.

The behavior of a process is specified as a series of actions that occur over time. Relationships within or between processes may be time-dependent or dependent upon the state of the model. As several processes may be taking place simultaneously, interaction may occur. Complex structural relationships, involving several types of entities, may be formed.

The actions of a process change a system state in four ways:

1. They modify the state of an entity by changing the values of its attributes.
2. They expand or contract the model by creating and destroying entities.
3. They rearrange the model by moving entities in and out of sets.
4. They effect the interactions in the model within and between processes.

In addition, a process makes decisions to determine what changes in the model to effect.

Briefly stated, system representation entails specifying each type of entity that will exist while the model is operating. In addition, for each type of process a behavior pattern is specified. Each behavior pattern describes a series of actions that occur over a period of time. When the model is operating, the combined effect of the specified actions is to simulate the changes of state that occur while a system is in operation.

3.2. EXPERIMENTATION. The specification of a model is entirely descriptive in character. When the model has been completed, the description of component types and behavior is complete, but the model itself does not exist within the computer, nor is it operating.

Before a simulation study can be conducted, a detailed description of the experiment is required. All the facilities of the language which are used in the specification of a model may also be employed in the specification of an experiment. Processes may be described whose behavior have significance only with respect to a particular experimental environment.

3.2.1. *Experimental Conditions.* Experimental conditions must be enumerated for each run of the specified experiment. These test conditions constitute those controllable factors which are subject to variation between runs. Depending upon the scope of the data, they may be specified within the description of the experiment, or input from external sources during the run.

3.2.2. *Initialization.* The establishment of an initial model state is an important factor in any experimental run. The model must be populated with the desired number of entities of each type, and appropriate attribute values must be designated for each entity so created. The full initialization of the model includes the establishment of any sets that are initially required, and the starting of at least one model process.

For the first time the model may be said to exist. It is now in its initial state and will perform in response to the

state changes induced by processes of the various types defined. The model may be entirely self-sustaining once an initial model process is started, or it may be intermittently self-sustaining with control going back to the experimental process from time to time. In either case, at least one process must be started to initiate the conduct of the experiment.

3.2.3. *Observation.* The conduct of the experiment, that is, the running of the model under specified experimental conditions, is without significance unless the performance of the model is observed and recorded. The process concept is ideally suited to the specification and execution of observational processes. Each type of observation will specify the entity or entities to be observed and the conditions under which measurements are to be taken.

3.2.4. *Analysis and Output.* Observed data may be either retained with the simulation or placed on external storage for subsequent analysis and processing, depending upon the volume of data collected. In either event, processes may be defined for analyzing the data and providing reports to the user.

A central feature of the language is that there is no distinction between facilities for describing the system to be simulated and facilities for describing the experiment to be conducted with that model. In both instances the unit of description is the process. For each type of process a characteristic behavior pattern is specified, and as each process occurs its characteristic behavior pattern is executed. Flexible referencing makes possible significant variations between the behavior of processes of the same type, and it facilitates the construction of generalized modules. Processes may be specified exclusively for experimental purposes without affecting either the description or the performance of the model.

4. A Sample Simulation Study

The study of a simple problem by the use of simulation will be used to help clarify and illustrate some of the features of the language referred to in Section 3. The problem concerns the relationship between decision rules employed in the sequencing of orders and the resulting load on specific production facilities within a steel mill. This is clearly only a single aspect in the evaluation of decision rules. However, it is a problem that might validly be studied by simulation, since a relatively straightforward experiment can be defined for it. More complex problems would tend to require more complex experimental processes, thereby utilizing the capabilities of the language more completely.

The model consists of a segment of a steel mill starting from the open hearth furnaces and continuing through a primary rolling mill operation. The experiment consists of a simple block design evaluating three alternative sequencing rules over a range of three levels of order loads on the steel mill (Figure 1).

It is not the purpose of this paper to define or explain the statements of the language in detail. The example is

MEAN INPUT LOAD, mill orders/week
Level 1 = 100 orders/week
Level 2 = 200 orders/week
Level 3 = 300 orders/week
ORDER SEQUENCING RULE
Standard
Priority
Weighted
RESPONSE Mean contents of soaking pit

Mean input load (orders/week)	Order sequencing rule		
	Standard	Priority	Weighted
100	Run 1	Run 2	Run 3
200	Run 4	Run 5	Run 6
300	Run 7	Run 8	Run 9

BLOCK DESIGN

FIG. 1. Sample experiment

intended solely to serve as a vehicle for providing an overview and for pointing out some of the more interesting features of the language. In addition, selected portions of the language are listed in the Appendix.

4.1. A MODEL. The system to be described consists of the initial processes in the making of steel, the facilities involved in carrying out these processes, the materials on which the processes are performed, and the mill orders used to control the flow of materials through the mill.

The system operates as follows.

1. The furnaces are charged with raw material and produce molten steel.
2. The molten steel is poured into molds.
3. The steel hardens in the molds to form ingots that are then heat treated in a facility called the "soaking pit."
4. The heat-treated ingots are then rolled into slabs.

4.1.1. *Component Description.* Entity types that will be considered in this model of a steel mill are specified formally in the component description section of the model (Figure 2). The identifiers assigned to the entity types are followed by a colon, which indicates that the specification of the entity structure is to follow. This structure is defined by identifying attributes that are to be associated with each entity of this type. For example, the attribute WEIGHT will be associated with each INGOT. Each specification is terminated by a semicolon. The values of these attributes collectively represent the status of a particular entity at a point in time, whereas the number of attributes that are associated with any entity is determined by the level of detail to which that entity is being modeled. The modes of values that may be assigned to attributes are: numeric, literal character strings, or the name of an entity. Modes need not be specified explicitly; if not, they are established automatically. The ability to manipulate the name of an entity as a variable permits the formation of complex structural relationships that may be modified dynamically.

```
STEEL_MILL:        MODEL;
FURNACE:           STATUS;
INGOT:             WEIGHT;
SOAKING_PIT:       STATUS, CONTENTS;
MILL_ORDER:        DUE-DATE, QUANTITY, PRODUCT, GRADE;
CHARGE:            PROCESS;
STEEL_MAKING:      PROCESS PROCESS_TIME, ORDER_SEQUENCING, FURNACE;
SOAK:              PROCESS;
ORDER_GENERATION:  PROCESS BACKLOG, LOAD;
STANDARD_SEQ:      PROCESS;
PRIORITY_SEQ:      PROCESS;
WEIGHTED_SEQ:      PROCESS;
END;
```

FIG. 2. Component description section of a STEEL MILL model

The specifications of types of entities that contain the keyword, *PROCESS*, indicate that these entities possess dynamic characteristics which will be specified subsequently in the behavior description section of the model.

4.1.2. *Behavior Patterns.* The operation of the steel mill is described by specifying a behavior pattern for each type of process. A behavior pattern is described by using the statements of the language. These include control statements for specifying relationships within a process and interactions between processes.

The behavior pattern for ORDER_GENERATION, a simple process-type whose purpose is to introduce MILL_ORDERs into the simulated steel mill, is shown in Figure 3.

```
ORDER_GENERATION:   PROCESS        LOCAL     ORDER;
            L1:     CREATE   MILL_ORDER   NAMED     ORDER;
                    ORDER.GRADE = GRADE_FUNCT(LOAD);
                    FILE ORDER IN BACKLOG;
                    TAKE INTERARRIVAL (LOAD);
                    GO TO L1;
END ORDER_GENERATION;
```

FIG. 3. Behavior description—ORDER_GENERATION process.

The order generation process represents the influence of processes that are external to the portion of the steel mill that is being represented and would probably be a fairly complicated process. For illustration, it will be sufficient to think of order generation in a very simple way. The process can be assumed to involve the creation of a single mill order to which the grade of material to be produced is assigned before it is filed in a SET of orders named BACKLOG.

To discuss order generation, we will assume that a SET has been created and that the name of that SET has been assigned as the value of the attribute, BACKLOG, at the time that an ORDER_GENERATION process was started.

A local attribute ORDER is defined for use within the behavior block. Each statement (step in the process) is terminated by a semicolon. As with entity specifications, a free-field format is used for writing and punching; statement labels are optional.

The *CREATE* statement enters a MILL_ORDER into the model and assigns its name as the value of ORDER. This name is then used, in the next statement, to assign a value to the GRADE attribute of that MILL_ORDER. In addition to identifying an entity, a name value indicates the type of that entity; thus, it is possible to ask if ORDER is a MILL_ORDER. A name may be assigned as the value of any attribute. In the ORDER_GENERATION process the attribute BACKLOG has as its value the name of a SET; it is used in the *FILE* statement to identify the particular SET in which the MILL_ORDER is to be placed. The statement records the name of the MILL_ORDER, given by ORDER, in the SET named BACKLOG. Other statements allow for selecting subsets of entities in a set, based on logical conditions, and executing one or more statements each time an entity is selected.

The *TAKE* statement specifies the length of simulated time between the arrival of orders at the steel mill.

The STEEL_MAKING process illustrates a somewhat more complex behavior pattern. The process of making steel from raw materials in a single furnace is a continuing process; i.e., once started, it continues indefinitely until it is stopped. While steel making is taking place it interacts with other processes in various ways; e.g., it uses the charge process as a subprocess and it controls the start of the order sequencing and soak processes. Once started, the latter two processes take place concurrently with the steel making process.

Assuming that the steel making process has been started with a particular furnace, the status of the furnace is checked to see whether it is available, and if so, the charge operation is performed. The order sequencing process selects the orders to be applied to the output of this furnace. Steel making is simulated by taking processing time. When this is complete, heat treatment of the group of ingots that has been produced begins by starting the soak process. Then the status of the furnace is again checked. The specification of this behavior pattern is shown in Figure 4.

The steel making process has three attributes: PROCESS_TIME, ORDER_SEQUENCING, and FUR-

```
STEEL_MAKING:  PROCESS

        S1:  IF FURNACE.STATUS EQ 0

             THEN BEGIN

                  WAIT CHANGE (FURNACE.STATUS);

                  GO TO S1;

             END ;

     PERFORM CHARGE;

     START ORDER_SEQUENCING;

     TAKE PROCESS_TIME;

     START SOAK;

     GO TO S1;

END STEEL_MAKING;
```

FIG. 4. Behavior description—STEEL_MAKING process

NACE, which were assigned when the steel making process was started. (See Initialization, Figure 7.) The statement labeled *S1* tests the status of the furnace by interrogating the value of its attribute, STATUS. The *WAIT CHANGE* statement will cause the steel making process to be suspended until a new value has been assigned to FURNACE. STATUS. At this time, the *GO TO S1* statement will be executed. The *PERFORM* CHARGE statement initiates a charge process and causes the steel making process to be suspended until that process has been completed. The *START* ORDER_SEQUENCING statement initiates an order sequencing process to be run in parallel with the steel making process. The value of the ORDER_SEQUENCING attribute is the name of a type of process which will be a decision rule, e.g., STANDARD_SEQ. The *TAKE* PROCESS_TIME statement simulates the making of steel from raw materials.

4.2. AN EXPERIMENT. The block design of the experiment shown in Figure 1 requires nine runs covering all possible combinations of mean input loads and order sequencing rules. The performance measure for this experiment will be a statistic concerning the contents of the soaking pit, e.g., the average number of ingots in the soaking pit during the run, or the maximum number of ingots in the pit at any one time. Mill orders entering the system are sampled from a stationary distribution whose mean is the mean input load of the run. Observations of the contents of the soaking pit will be made whenever the contents change, in order to generate the necessary response statistic.

The formal description of the experiment requires the definition of experimental entities and processes. However, no explicit distinction need be made between entities employed within the model and entities used only for purposes of experimentation. The experimental entities and processes for the steel mill study are shown in Figure 5. These would appear in the component description section (Figure 2). The RUN_CONDITION entity carries test data and results unique to each run of the experiment. The EXPERIMENT, INITIALIZATION and OBSERVATION processes are discussed below.

```
RUN_CONDITION:     LOAD, RULE, RESPONSE, TIME ;

EXPERIMENT:        PROCESS;

INITIALIZATION:    PROCESS RUN;

OBSERVATION:       PROCESS QUEUE, WEIGHTED_SUM, LAST_CHANGE,
                       LAST_CONTENTS, MEAN;

CLEAR:             PROCESS;

RUN_ANALYSIS:      PROCESS RUN;

FINAL_ANALYSIS:    PROCESS;
```

FIG. 5. Experimental entities

In the steel mill study the EXPERIMENT process (Figure 6) controls the entire experiment, providing for initialization at the start of each run and data analysis following each run, as well as at the end of the experiment. In this case only one such process is required. It is started at the outset and continues throughout the simulation, finally causing the end of the simulation by the statement *TERMINATE MODEL*.

INITIALIZATION (Figure 7) provides the means for establishing a reasonable initial state for an experimental run, including the processes which are active at the start of the run. The conditions for the run are read in, including a process-type (RUN.RULE) as well as numeric data such as RUN.TIME. In this manner it is possible to control the type of activity undertaken by the model in response to information read in during simulation. Initialization of the ORDER_GENERATION process provides necessary data to that process and also *CREATES* a SET (BACKLOG) which will henceforth be associated with it. The number of FURNACEs is fixed for the duration of the run. So they are all created at once and a STEEL_MAKING process is started on each FURNACE. This is accomplished with the *FOR EACH* statement which provides for repetition of the statement following the *DO* over all or any desired subset of the FURNACEs. Note that the value assigned to the ORDER_SEQUENCING attribute of STEEL_MAKING is the type of decision (process) to be tested in this run. The desired decision rule will be called during STEEL_MAKING by the *START* ORDER_SEQUENCING statement (see Figure 4). Hence it is possible for different STEEL_MAKING processes to employ different decision rules, although they share the same behavior description.

The final process to be initialized is the OBSERVA-

TION process. This is an "experimental" process, rather than a "model" process, but one such process will be utilized for each run. The name of the OBSERVATION process is assigned to the RESPONSE attribute of a RUN_CONDITION to facilitate maintaining data on the results of the run, within the process. Although not necessary, it is a convenient option in this case. The OBSERVATION process, described in Figure 8, collects data concerning model behavior without altering the behavior description in any way. This is accomplished through the use of automatic signaling (as specified by the *WAIT CHANGE* statement). Every time an INGOT is added to or removed from the SOAKING_PIT, a signal is received by the OBSERVATION process. At these times the appropriate statistics are updated, and the OBSERVATION process again reverts to a waiting status. This process has also been generalized to gather queue

```
EXPERIMENT: PROCESS LOCAL R, I, #_RUNS;

        /* READ RUN LENGTH */

    GET SYSIN (#_RUNS);

        /* EXECUTE THE EXPERIMENT */

    FOR I = 1 TO #_RUNS DO

        BEGIN

            CREATE RUN_CONDITION NAMED R;

            PERFORM INITIALIZATION WHERE RUN = R;

            TAKE R.TIME;

            PERFORM RUN_ANALYSIS WHERE RUN = R;

            PERFORM CLEAR;

        END;

    PERFORM FINAL_ANALYSIS;

    TERMINATE MODEL;

END EXPERIMENT;
```

FIG. 6. Behavior description—EXPERIMENT process

statistics on any SET. Many such observations could, of course, be conducted in parallel.

Figure 9 summarizes the format of the source code for the STEEL_MILL model. The *EXECUTE* statement, following the behavior descriptions, specifies the first process to be started. The data is supplied as required by the experiment outlined in Figure 1, and is read in by the *GET* statements in the experiment and initialization processes.

5. Highlights of the Language

5.1. DATA STRUCTURES. There are three modes of data. The value of an attribute may be a character, a number, or the name of an entity. Values may be assigned to

```
INITIALIZATION: PROCESS
        LOCAL N, B, F, PIT ;

            /* READ RUN CONDITIONS */
        GET SYSIN (RUN.LOAD, RUN.RULE, RUN.TIME, N) ;

        /* INITIALIZE ORDER GENERATION PROCESS */
        CREATE SET NAMED B ;
        START ORDER_GENERATION
            WHERE LOAD = RUN.LOAD, BACKLOG = B ;

        /* INITIALIZE STEELMAKING PROCESS */
        CREATE ALL N OF FURNACE;
        FOR EACH F OF FURNACE DO
            BEGIN F.STATUS = 1;
                START STEEL_MAKING WHERE
                    PROCESS_TIME = 8,
                    FURNACE = F,
                    ORDER_SEQUENCING = RUN.RULE ;
            END ;

        /* INITIALIZE OBSERVATION PROCESS */
        CREATE SOAKING_PIT NAMED PIT;
        CREATE SET NAMED PIT.CONTENTS;
        START OBSERVATION NAMED RUN.RESPONSE WHERE
            QUEUE = PIT.CONTENTS ;

END INITIALIZATION ;
```

FIG. 7. Behavior description—INITIALIZATION process

variables. A variable is a single-valued attribute or an element in an array. A group of attributes is associated directly with the model, which itself can be viewed as an entity. When an entity is created it is assigned a name which both identifies it and indicates what type of entity it is. Relationships between entities are established by assigning the name of one as the value of an attribute of another.

The ability to create and destroy entities and to manipulate their names permits complex relationships to be formed dynamically. Observation and maintenance of interrelated entities is facilitated by the ability to ask about the nature of an entity.

The *FILE* and *REMOVE* statements and such generic expressions as *FIRST*, *LAST*, and *ANY* provide a flexible facility for constructing and utilizing SETs of entities.

5.2. PROGRAM STRUCTURES. Each statement represents a step in the behavior pattern of a process. A block of statements may be combined to function as a single statement in two ways: first, by delimiting the block by *BEGIN...END*; and second, by delimiting the block by *FUNCTION...END*;. In either case, local variables may be defined for use within the block of statements. A *BEGIN* block is executed in line, while a *FUNCTION* may be called from many places, each time with different parameters. The name of a *FUNCTION* may be manipulated in the same manner as any entity name. A *FUNCTION* is called by referencing a variable whose value is the name of that *FUNCTION*. For example, consider the following sequence of statements, where *PR*1 and

```
OBSERVATION:  PROCESS

        L1:   WAIT CHANGE (QUEUE.LIST_COUNT);

              WEIGHTED_SUM = WEIGHTED_SUM + LAST_CONTENTS *
                             (CLOCK - LAST_CHANGE);

              LAST_CHANGE  =  CLOCK;

              LAST_CONTENTS = QUEUE.LIST_COUNT;

              GO TO L1;

END OBSERVATION;
```

FIG. 8. Behavior description—OBSERVATION process

```
STEEL_MILL:   MODEL;

              /*  COMPONENT DESCRIPTIONS  */

END ;

              /*  BEHAVIOR DESCRIPTIONS  */

EXECUTE EXPERIMENT ;

                        /*  DATA  */

/*  #_RUNS  */   9

/*RUN_#        LOAD      RULE              TIME      NUMBER FURNACES  */
/*  1 */        100       STANDARD_SEQ      400       5
/*  2 */        100       PRIORITY_SEQ      400       5
/*  3 */        100       WEIGHTED_SEQ      400       5
/*  4 */        200       STANDARD_SEQ      400       5
/*  5 */        200       PRIORITY_SEQ      400       5
/*  6 */        200       WEIGHTED_SEQ      400       5
/*  7 */        300       STANDARD_SEQ      400       5
/*  8 */        300       PRIORITY_SEQ      400       5
/*  9 */        300       WEIGHTED_SEQ      400       5
```

FIG. 9. Summary of STEEL MILL model

PR2 are *FUNCTION*s:

```
      ⋮
TEST A EQ B
  THEN X = PR1;
  ELSE X = PR2;
S2: K = X(ARG);
      ⋮
```

The *FUNCTION* that is called by statement *S2* is determined by the outcome of the test. *FUNCTION*s may be recursive. *FUNCTION* arguments may be an expression or a reference to any attribute.

Each process behavior description is a sequence of statements which may include *BEGIN* blocks and function calls. A process may interrupt itself until certain conditions exist, or it may delay the completion of another process in the system. One process may perform another process as a subprocess, or it may start other processes that will operate simultaneously. Many processes of the same type may be taking place at the same time, i.e., following the same behavior pattern, but they need not be synchronized.

5.3. CONTROL OF PROCESSES. The control statements provide a means for specifying the relationships between actions in a process and relationships between two or more processes. Two statements, *START* and *PERFORM*, initiate a process and establish its relationship with the initiating process. The *START* statement causes the initiating and initiated processes to proceed concurrently. The *PERFORM* statement causes the initiating process to suspend further action until the performed process has terminated. These statements permit the assignment of values to any or all of the attributes of the process being initiated. While the process exists in the model, its attributes may be referenced by any other process that has access to its name. A *WAIT* statement can interrupt a process until one or more of a list of attributes changes, or until one or more of a list of processes has terminated. Further, *SUSPEND* and *RELEASE* statements can be used by one process to control the progress of another, and the *TERMINATE* statement can be used to terminate either the issuing process, another process, or the experiment. The *TAKE* statement simulates processing time.

5.4. OBSERVATION. The concept of a process that exists over time greatly facilitates the gathering of data about an experiment. Observation processes can be written to gather data periodically or when certain changes of state occur. Automatic signaling, specified by a *WAIT CHANGE* or *WAIT END* statement, permits the construction of an observation process independently of the process being observed.

5.5. INPUT/OUTPUT. External storage is utilized for input files, output files, files of entities, and files of programs. The input and output files are regarded as a continuous stream of fields. Each field may be an integer, a real number, an entity type name, or a character string. Each auxiliary file may contain entities of one type. Program files may contain the behavior pattern for any process or function.

5.6. LANGUAGE EXTENSIONS. The language may be extended through the use of a macro facility that permits the definition of new statements and expressions in terms of those already defined (see [10]). Some of the statements that have been discussed (*START*, *PERFORM*) and some generic expressions (*FIRST*, *LAST*, *MEAN*, *VARIANCE*) are macros that are defined in terms of other statements and expressions. For example, in the statement,

```
ORDER = FIRST ORDER OF BACKLOG WITH ORDER.
        GRADE EQ 2;
```

the expression following "=" is an expression macro, identified by *FIRST*, that is defined in terms of a *FOR EACH* statement. The function of the macro is to select the first entity in a SET that satisfies the specified condition, with the name of the selected entity being the value of the expression.

A macro is specified in two parts: (1) by the macro structure, which describes the syntax of the new statement or expression in terms of a set of syntatic units, e.g., variable or expression, and (2) by the macro definition, which describes the semantics of the new statement or expression in terms of statements in the language or previously defined macros.

6. Summary

This paper has attempted to identify the major elements of a simulation study and to indicate the manner in which the language deals with these elements. The language has not been described in detail, but some of its features have been illustrated by a simple example. The major orientation of the language is toward:

(1) A unified approach to simulation in which the facilities of the language are equally suitable for the description of a system and for the description of an experiment to be carried out upon that model.

(2) A "world view" for system description which explicitly embodies the notion of process for describing dynamic behavior. The process is seen as an ideal vehicle for describing either systems behavior or experimental procedures. A process exists as an entity in the model while simulation is taking place. The process may directly change the state of an entity; it may create or destroy entities, freely rearrange the groupings (set memberships) of entities, specify interactions within and between processes in either a time-dependent or a state-dependent manner, and it may execute complex logical decisions.

(3) A modular capability for modeling and experimentation. The ability of a process to exist within the model for any desired period of simulated time makes possible the construction of behavior modules whose scope can be dictated entirely by the requirements of the problem. Systematic attacks on particular problems or classes of problems can be mounted by building general purpose modules to serve as the building blocks for a variety of models and experiments.

It is premature to assess fully the utility of the language features discussed in this paper. Further study of the language and of its experimental implementation is planned.

Acknowledgment. The language described in this paper was developed at the Advanced Systems Development Division of IBM. The group included K. R. Blake, B. M. Leavenworth, and S. C. Pierce, ASDD; and G. P. Blunden, IBM United Kingdom (presently with CEIR Ltd.).

RECEIVED JANUARY, 1966; REVISED JULY, 1966

APPENDIX

The following is a description of selected portions of the language. The notation used in this description is:

1. Lower-case letters are used for metavariables.
2. [ ] are used to enclose options.
3. { } indicate that one of the enclosed constructions must be used.

A MODEL

```
/* COMPONENT DESCRIPTION SECTION */
   [identifier:] MODEL
  [model-attribute-declarations];
    entity-type-declarations
      END [identifier]  ;
  /* BEHAVIOR DESCRIPTION SECTION */
    behavior-descriptions
    EXECUTE process-name  ;
```

COMPONENT DESCRIPTION SECTION

Attribute-Declaration

identifier [<dimensions>] [{ [/size/] *INTEGER* / *REAL* / *NAME* / *CHAR* }]

Entity-Type-Declaration

identifier: attribute-declarations ;
identifier: *PROCESS* [attribute-declarations] ;

BEHAVIOR DESCRIPTION SECTION

Behavior-Description

identifier: *PROCESS* [(parameter-list)] [;]
[*LOCAL* attribute-declarations ;]
statement-list
END [identifier] ;

Statements

attribute-reference = expression ;
CREATE type-name [*NAMED* attribute] [*WHERE* assign-group] ;
DESTROY entity-name ;
FILE name *IN* set [{ *BEFORE* name / *AFTER* name / *AT HEAD* / *AT TAIL* }] ;
REMOVE name *FROM* set ;
FOR [*EACH*] repetitive-assignment *DO* statement
IF expression *THEN* statement
TEST expression *THEN* statement *ELSE* statement
START process-type [(argument-group)]
[*NAMED* attribute] [{ *AT* / *AFTER* } expression]
[*WHERE* assign-group] ;
PERFORM process-type [(argument-group)] [*NAMED* attribute]
[*WHERE* assign-group] ;
TAKE [*UNTIL*] expression ;
SUSPEND process-name [expression [*TIMES*]] ;
RELEASE process-name [{ expression / *ALL* } [*TIMES*]] ;
WAIT CHANGE [expression] (attribute-references) ;
WAIT END [expression] (name-attribute-references) ;
TERMINATE [{ process-name / *MODEL* }] ;

GET data-set (input-list) ;
PUT data-set (output-list) ;

BEGIN-END Block

BEGIN [;]
[*LOCAL* attribute-declarations ;]
statement-list
END [identifier] ;

FUNCTION Description

identifier: *FUNCTION* [(parameter-list)] [;]
[*LOCAL* attribute-declarations ;]
statement-list
END [identifier] ;

REFERENCES

1. BLAKE, K. R., AND GORDON, G. Systems simulation with digital computers. *IBM Sys. J. 3*, 1, (1964), 14.
2. BLUNDEN, G. P., AND KRASNOW, H. S. The process concept as a basis for simulation modeling. Paper, 28th Nat. Meeting ORSA, Houston, Texas, Nov. 4–5, 1965.
3. BUXTON, J. N., AND LASKI, J. G. Control and simulation language. *Comput. J. 5*, 3 (Oct. 1962), 194–199.
4. DAHL, O., AND NYGAARD, K. SIMULA—an ALGOL-based simulation language. *Comm. ACM 9*, 9 (Sept., 1966), 671–678.
5. EFRON, R., AND GORDON, G. A general purpose digital simulator and examples of its application: Part I—description of the simulator. *IBM Sys. J. 3*, 1 (1964), 22.
6. KNUTH, D. E., AND MCNELEY, J. L. SOL—A symbolic language for general purpose systems simulation. *IEEE Trans., EC-13*, 4 (Aug. 1964), 401–408.
7. KRASNOW, H. S. Dynamic representation in discrete interaction simulation languages. *Digital Simulation in Operational Research*. The English Universities Press Ltd., London, 1967, pp. 77–92.
8. KRASNOW, H. S. Highlights of a dynamic system description language. Paper, 29th Nat. Meeting ORSA, Los Angeles, Calif., May 1966.
9. KRASNOW, H. S., AND MERIKALLIO, R. A. The past, present and future of general simulation languages. *Man. Sci. 11*, 2 (Nov. 1964), 236–267.
10. LEAVENWORTH, B. M. Syntax macros and extended translation. *Comm. ACM 9*, 11 (Nov. 1966), 790–793.
11. LEAVENWORTH, B. M., AND PARENTE, R. J. Structure of sequencing algorithms in simulation languages. *Information Processing 1965*; Proc. IFIP Congress 65, Vol. 2, Spartan Books, Washington, D.C., 1965.
12. MARKOWITZ, H. M., ET AL. *SIMSCRIPT, A Simulation Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1963.
13. TOCHER, K. D. A review of simulation languages. *Oper. Res. Quart. 16*, 2 (June 1965), 189–217.
14. TOCHER, K. D., AND HOPKINS, D. A. Handbook of the General Simulation Program, Mk. II. United Steel Companies Ltd., Sheffield, England, June 1964.

An Algorithm for Class Scheduling With Section Preference

VINCENT A. BUSAM*

Computer Sciences Corp., El Segundo, Calif.

An algorithm for assignment of students to classes in a fixed time schedule that allows students to give a preference for sections within courses is given. If consistent with the objective of balanced sections, these preferences will be honored. The algorithm is more stochastic than Monte Carlo in nature. Results are given that compare it to a nonpreference assignment algorithm.

1. Introduction

Various attempts have been made at developing programs which will schedule students in classes [1–3]. The common goal seems to be, given the students' requests for classes, to schedule them in sections such that: (1) students are given a nonconflicting class schedule, if one is available, and (2) sections are filled as evenly as possible.

Algorithms used have achieved the objective fairly well. With one exception [3], programs schedule students in classes based only on course requests and do not allow section preference. This is a common criticism of machine sectioning, since the student loses all chance of choosing a convenient time and/or room location schedule or of picking sections taught by favorite instructors.

This paper describes an algorithm that allows a large degree of section preference while maintaining the original two objectives of machine registration. This algorithm was tested using the entire registration for one semester at Washington State University. A comparison of the results achieved by a nonpreference algorithm and the preference algorithm is given.

2. Nonpreference Alogrithm

At Washington State University, the student fills out one request card for each class he wishes to take. No alternate choices are allowed except for Physical Education (PE) classes, where two alternates are also given. A general description of the data collection procedure and conflict resolution used is given by Faulkner [1].

The algorithm currently used at WSU orders each student's courses according to the number of sections still

* Research performed at Washington State University, Pullman, Washington