1990 EMC Abstract

Model Instantiation Algorithms for Query Driven Simulation in SIMODULA

John A. Miller and Orville R. Weyrich, Jr.
Department of Computer Science
University of Georgia

Query driven simulation is a powerful paradigm for simulation modeling and analysis. We are currently in the process of completing our first prototype implementation of an environment supporting this paradigm. This prototype is called the *SIMODULA/OBJECTR simulation support environment*. The eventual goal is to have an integrated data, knowledge and model base. Users may interact with the system at several level of sophistication (i.e., from the entry of simple queries to the construction of complex models directly in SIMODULA). [SIMODULA is a simple modular *process-oriented simulation language*.] The unifying theme, however, is that most interactions with the environment will be in the form of queries.

Queries formulated in an extended SQL-like query language, allow users to retrieve information about the behavior of systems under study. Data from simulation runs is stored by OBJECTR, which is a *relational database system with object-oriented extensions*. If the information requested by the query is already stored in the database, more or less conventional query processing techniques will be applied. However, if the amount of information retrievable is below a user-settable threshold, model instantiation and execution will be automatically carried out, to generate the desired information.

Model instantiation involves the creation of model instances that can then be executed. In its most elementary form, model instances are created by parameterizing a simulation model with values formed by schema and query analysis. Our model instantiation algorithm, implements this capability by transforming the WHERE clause of the query to disjunctive normal form. Each of the disjuncts (ORed terms) is used to create a single model instance. [Note, on the appropriate hardware (e.g., a multiprocessor) each model instance could be executed in parallel.] The next level of complexity occurs when the query references multiple models. This capability allows alternative systems to be compared. Common characteristics of the alternative systems, may be matched up by specifying join conditions in the query.

We conclude by discussing how OBJECTR could used for model synthesis. Since modeling elements could be stored in a model base, a query could return modeling elements, or indeed an entire simulation model as its result. This powerful facility, would give users the capability to browse a model base, seeing what is available. If the appropriate modeling elements have already been coded, then the user could very rapidly create his own simulation models, simply by formulating the appropriate query.

1. Introduction

Simulation modeling is a process of mapping inputs to outputs. Performing a complete study of a complex system can be an enormous task. With a vast array of scenarios and input parameter settings, and with a correspondingly large amount output data, the management of such becomes quite difficult. Additionally, many parameter settings may be very detailed and difficult for a modeler to remember (e.g., 23 millisecond mean access time to a disk).

In this paper we consider a particular approach to simplifying this problem. We have developed a simulation system that consists of a *process-oriented simulation language*, called SIMODULA, and a *relational database system with object-oriented extensions*, called OBJECTR.

SIMODULA - SImulation MODULAr
OBJECTR - OBJECT Relational

Both are written in Modula-2. SIMODULA was developed as a language to teach simulation. The language is small enough to be very easily learned by Pascal or Modula-2 programmers. However, it is powerful enough to do complex simulations. Indeed it provides most of the process-oriented features of SIMSCRIPT II.5. Additionally, because of its compatibility with a subset of SIMSCRIPT II.5, it can be used as a stepping stone to this very powerful simulation language.

SIMODULA and OBJECTR are interfaced to provide a simple yet powerful simulation modeling environment. When a modeler is studying a system, he wants to know what happens given a scenario. Thus he is asking a question that he wants an answer to. Our approach uses this framework for simulation modeling. A modeler simply queries the system using an SQL-like query language. If the information already exists in the database, an ordinary retrieval is done to answer the query. However, if the information does not exist, it must be generated. Our system automatically generates the information by *instantiating simulation models* to create information about the system.

2. Simulation Using SIMODULA

SIMODULA is an easy to learn process-oriented simulation language written in Modula-2. The *module* and *coroutine* features of Modula-2 make it well suited for implementing simulation languages (or rather extensions to Modula-2 using library modules). The coroutine feature of Modula-2, although low level, provides what is needed to implement processes for simulation models. Essentially a process runs until it needs to either perform an activity or wait on a resource. At such a time the coroutine can give up execution, allowing some other coroutine to run. The coroutine chosen to run is the process that should be run next in the simulation. This is accomplished by maintaining *pointers to process descriptors* (ProcessPtr) on a Future Event List, and using a TRANSFER statement to cause a context switch. The address of the coroutine is one of the process descriptor's fields. The complexity of both the Future Event List and the coroutine transfers is hidden from the user by being encapsulated in a module.

The module *simulate* provides the basic capabilities to do process-oriented simulation. An early version of the definition module [Mill 88], simulate.def, evolved from the definition module given in [Hugh 87]. This module has been enhanced to include most of the process-oriented features of SIMSCRIPT II.5. Indeed it is compatible enough to facilitate semi-automatic translation between SIMODULA and SIMSCRIPT II.5. The simulate module provides procedures to CREATE, ACTIVATE, WORK, SUSPEND, INTERRUPT, RESUME, and DESTROY processes. They can be viewed by users of the module as language extensions, a simulation language built on top of Modula-2. Once a process is CREATEd, it can be ACTIVATEd or scheduled for execution by placing it on the Future Event List according to its future execution time and its priority. A process may delay itself for a definite wait by executing the WORK statement, or may delay itself for an indefinite wait by executing the SUSPEND statement. A process may INTERRUPT another process in the middle of WORK and later RESUME that process. Finally, when a process completes, the DESTROY procedure is called causing a coroutine transfer to occur. Note, it is necessary to place a DESTROY at the end of coroutines, since falling off the end of a coroutine generates a program halt. To initiate a simulation the procedure StartSimulation is called. In addition to the above fundamentally important procedures several support procedures are also provided by the simulate module. Many are in the form of procedures or functions that retrieve information encapsulated in the simulate module. This safety feature allows the information to be read but not written to. The simulate definition module is shown below. It describes each procedure and also mentions the equivalent SIMSCRIPT II.5 features.

```
DEFINITION MODULE simulate; (*
        This module provides the basic capabilities to do process-oriented simulations.  It allows
        one to create quasi-concurrent processes which evolve through time by performing a se-
        quence of activities. The processes are scheduled for execution on a hidden Future Event
        List.  The possible states that a process can be in are: PASSIVE - waiting indefinitely for
        some other process to activate it; ACTIVE - in the middle of some activity - on the Fu-
        ture Event List; SUPERACTIVE - the currently executing process; INTERRUPTED - in-
        terrupted in the middle of some activity; and DESTROYED - the process no longer exists
        and only a pointer to it remains. *)

FROM SYSTEM IMPORT WORD;

CONST
        NOW = 0.0;

TYPE
        STATUS  = (PASSIVE, ACTIVE, SUPERACTIVE, INTERRUPTED, DESTROYED);
        ProcessPtr;  (* Opaque - pointer to process descriptor *)

PROCEDURE CREATE(VAR Process: ProcessPtr; Procedure: PROC; Attrib: WORD); (*
        Create a new Process whose specification is given by the parameter Procedure.  Note,
        procedures of type PROC have no parameters.  The descriptor for the process will in-
        clude a coroutine.  A new coroutine is created but control is not transferred to it. The
        parameter Attrib is a pointer to this process's attributes.
```

SIMSCRIPT: CREATE A Process; STATE TRANSITION: "" -> PASSIVE *)

PROCEDURE ACTIVATE(Process: ProcessPtr; TimeDelay: LONGREAL); (*

Activate or schedule a Process for execution. The process is inserted into the Future Event List according to first its activation time, and second its priority. The process's activation time is set to CurrentTime + TimeDelay. To activate a process to execute immediately use the constant NOW.

SIMSCRIPT: ACTIVATE THIS Process; STATE TRANSITION: PASSIVE -> ACTIVE *)

PROCEDURE WORK(TimeDelay: LONGREAL); (*

Delay the current superactive process until the clock advances to CurrentTime + TimeDelay. This is achieved by inserting the process into the Future Event List according to first its activation time, and second its priority, and then transferring to the process at the front of the Future Event List.

SIMSCRIPT: WORK TimeDelay UNITS; STATE TRANSITION: SUPERACTIVE -> ACTIVE *)

PROCEDURE SUSPEND; (*

Suspend (deactivate) the current superactive process. The first process in the Future Event List will be the next to be executed.

SIMSCRIPT: SUSPEND; STATE TRANSITION: SUPERACTIVE -> PASSIVE *)

PROCEDURE INTERRUPT(Process: ProcessPtr); (*

Interrupt (deactivate) a Process. This procedure allows a process to interrupt some other process.

SIMSCRIPT: INTERRUPT THIS Process; STATE TRANSITION: ACTIVE -> INTERRUPTED *)

PROCEDURE RESUME(Process: ProcessPtr); (*

Resume a previously interrupted Process. The process will be put on the Future Event List with a delay equal to its remaining time.

SIMSCRIPT: RESUME THIS Process; STATE TRANSITION: INTERRUPTED -> ACTIVE *)

PROCEDURE DESTROY; (*

Destroy or terminate the current superactive process. The first process in the Future Event List will be the next to be executed.

SIMSCRIPT: DESTROY THIS Process; STATE TRANSITION: SUPERACTIVE -> "" *)

PROCEDURE StartSimulation; (*

Start the simulation by removing the first process from the Future Event List and transferring to it.

SIMSCRIPT: Start Simulation *)

PROCEDURE ActivationTime(Process: ProcessPtr): LONGREAL; (*

```
        Return the activation time (the time it is scheduled for execution) of the Process.
        SIMSCRIPT: Time.A(Process) *)

PROCEDURE Priority(Process: ProcessPtr): CARDINAL; (*
        Return the priority of the Process where 0 is the highest priority.
        SIMSCRIPT: PriorityAttrib(Process) *)

PROCEDURE Status(Process: ProcessPtr): STATUS; (*
        Return the current status of the Process.
        SIMSCRIPT: Sta.A(Process) *)

PROCEDURE Attributes(Process: ProcessPtr; VAR Attrib: WORD); (*
        Return a pointer to the attributes of the Process in Attrib.
        SIMSCRIPT: AttribName(Process) *)

PROCEDURE Time(): LONGREAL; (*
        Return the current simulated time (i.e., the current value of the simulation clock, Current-
        Time).
        SIMSCRIPT: Time.V *)

PROCEDURE CurrentProcess(): ProcessPtr; (*
        Return the process currently executing (i.e., the superactive process).
        SIMSCRIPT: Process.V *)

PROCEDURE SetPriority(Process: ProcessPtr; ProcessPriority: CARDINAL); (*
        Set the ProcessPriority for the Process to other than its default value.  The highest priori-
        ty is 0 (the default value) while the lowest priority is MAX(CARDINAL).  Note, chang-
        ing the priority of an active process will not affect its position on the Future Event List.
        SIMSCRIPT: BREAK Process TIES BY LOW PriorityAttrib *)

PROCEDURE SetWorkspace(NewSize: CARDINAL); (*
        Set the coroutine WorkspaceSize to other than its default value. This procedure should be
        called before the relevant processes are created.
        SIMSCRIPT: handled automatically *)

END simulate.
```

There are four other modules in our Modula-2 simulation system. Source code for these definition modules is given in the Appendix. The module *resources* allows users to define resources. A resource has a number of service units fed by a common FCFS queue. A process acquires a unit of a resource by calling Request, and releases the unit by calling Relinquish. If a process Requests a unit of a resource and none are available, then the process waits in the resource's queue. The module *variates* provides procedures to generate random numbers and random variates. All the random variates provided by SIMSCRIPT II.5 are included in this module. The module *statistics* provides statistics gathering capabilities. Sample statistics can be gathered using the procedure Tally, while time-persistent statistics can be gathered using the procedure Accumulate. Note, statistics gathering is more automatic in SIMSCRIPT II.5, since explicit procedure calls need not be made. A low level module *lists* provides general list handling capabilities. It is used by the simulate module to manage the Future Event List, and by the resources module to manage its FCFS queues. It may also be employed by users to manage their queues. Note, in many cases the standard queuing facilities provided by the resources module will suffice so that users will not need to use the lists module directly.

As a simple example, we present the SIMODULA code for a classical bank simulation. [Abstractly this simulation model is just a M/M/s queue.] This bank simulation program consists of a teller resource and two process specification procedures. These procedures provide a script or template for created processes. The Generator process (the single process created from the Generator process specification procedure) is used to create Poisson arrivals of Customer processes. The Customer process specification procedure encodes a script for a typical customer. A customer requests a teller. If one is not available, the customer waits in the teller queue. Once the customer has a teller, he is served for an exponentially distributed amount of time. Finally, the customer relinquishes his teller and leaves the bank. [Note, Wirth's InOut I/O module is weak and inconvenient, so that we wrote our own (PascalIO) whose flavor is closer to I/O in Pascal. The letter after the word WRITE is the first letter of the data type your writing out.] After the program listing, output results generated from a sample run are shown. For this run the input parameters are set at their default values (see the next section).

```
MODULE Bank;        (* Bank Simulation Model *)

FROM Storage  IMPORT ALLOCATE, DEALLOCATE;
FROM PascalIO IMPORT WRITE, WRITEc, WRITEl, LN;
FROM simulate IMPORT CREATE, ACTIVATE, WORK, SUSPEND, DESTROY,
        StartSimulation, Attributes, CurrentProcess, Time, ProcessPtr, NOW;
FROM variates IMPORT Exponential;
FROM statistics IMPORT Tally, Mean, WriteStat, WriteLabels;
FROM resources  IMPORT ResourcePtr, CreateResource, Request, Relinquish,
        WriteResourceStat;
FROM bank IMPORT bankTup, bankRec, GetParameters, WriteTuple;

CONST TRACE = FALSE;

TYPE CustAttributes = POINTER TO RECORD
        CustNum: CARDINAL;
END (* CustAttributes *);

VAR Teller: ResourcePtr; Process: ProcessPtr; t: bankTup;

PROCEDURE Customer;
VAR arrivalTime, serviceTime, systemTime: LONGREAL; atr: CustAttributes;
BEGIN
        arrivalTime := Time();
        Attributes(CurrentProcess(), atr);
        IF TRACE THEN
                WRITEc(" Cust ", atr^.CustNum); WRITEl(" arrives ", arrivalTime); LN;
        END (* IF *);

        Request(Teller);
        serviceTime := Exponential(t^.MeanService, t^.Stream);
        Tally(1, serviceTime);
        WORK(serviceTime);
        Relinquish(Teller);

        systemTime := Time() - arrivalTime;
        Tally(2, systemTime);
        IF TRACE THEN
                WRITEc(" Cust ", atr^.CustNum); WRITEl(" has delay ", systemTime); LN;
        END (* IF *);
        DESTROY;
END Customer;
```

```
PROCEDURE Generator;
VAR
        i: CARDINAL; atr: CustAttributes;
BEGIN
        FOR i := 1 TO t^.NumCustomers DO
                WORK(Exponential(t^.MeanIArrival, t^.Stream));
                ALLOCATE(atr, SIZE(atr^)); atr^.CustNum := i;
                CREATE(Process, Customer, atr);
                ACTIVATE(Process, NOW);
        END (* FOR *);
        DESTROY;
END Generator;

PROCEDURE WriteResults;
BEGIN
        WITH t^ DO
                Throughput  := FLOAT(NumCustomers) / Time();
                MeanSysTime := Mean(2);
        END (* WITH *);
        WriteTuple(t);
        LN;
        WRITE(" Bank Simulation Results "); LN;
        LN;
        WRITEl(" Througput = ", t^.Throughput); LN;
        LN;
        WriteLabels;
        WriteResourceStat(Teller);    (* Utilization, Queue Length, Waiting Time *)
        WriteStat(1, "  Service Time");
        WriteStat(2, "  System Time");
END WriteResults;

BEGIN (* Bank *)
        ALLOCATE(t, SIZE(t^));
        GetParameters(t);
        CreateResource(Teller, t^.NumTellers);
        CREATE(Process, Generator, NIL);
        ACTIVATE(Process, NOW);
        StartSimulation;
        WriteResults;
END Bank.
```

```
 Bank Simulation Results

 Throughput = .110
```

Num	Obs	Min	Max	Mean	Variance	StdDev	Name
-3	912	0.000	1.000	0.844	0.131	0.363	Numb Service
-2	912	0.000	14.000	4.756	22.954	4.791	Queue Length
-1	100	0.000	111.127	43.400	1365.295	36.950	Waiting Time
1	100	0.038	31.689	7.705	41.081	6.409	Service Time
2	100	0.038	122.054	51.105	1377.061	37.109	Total Time

Briefly, these results show that the throughput is .110 customers per minute or 6.6 customers serviced per hour. The average time spent by customers in the bank is 51.105 minutes (maybe they should add another teller). [Note, this value is close to the value of 50.0 given by queuing equations.]

3. Simulation Support Using OBJECTR

3.1. OBJECTR: A Relational Database System with Object-Oriented Extensions

Relational database systems have been very successful in business applications. Information or data in such applications tends to be rather simple and fits naturally into a tabular structure. Information in relational databases is stored in the form of relations (tables), which are sets of tuples (rows). However, newer applications for database systems, such as engineering applications (CAD/CAM and VLSI databases) and expert system applications, demand greater functionality on the part of database systems. We believe that the structural properties of information associated with simulation modeling (e.g., input parameters, output results and model specifications) also call for enhancements to relational databases.

Currently much research is being conducted on extensions to relational database systems. Many of the proposed extensions involve adding abstract data type or object-oriented features to relational database systems. Such extensions are probably best illustrated by the extensions added to the INGRES relational database system. The first extension was to augment INGRES with abstract data types. This allowed more complex objects to be stored along with procedural information [Ston 86]. The second extension was to augment INGRES by allowing fields of a relation to be an arbitrary INGRES query [Ston 86]. A sequence of extensions to INGRES culminated in a project to entirely rewrite the database system [Ston 86b]. The new database system, called POSTGRES, includes some of the capabilities of knowledge base systems [Ston 86b]. Object-oriented approaches have already met with great success in software design [Booc 83] and programming languages (e.g., Simula 67 [Birt 73], Smalltalk 80 [Gold 83], C++ [Stro 86], and Lisp Flavors [Wein 81]). Modula-2, our development language, provides some support for object-oriented programming with its RECORD and MODULE features. However, it is weak in the sense that it does not directly support object types or object inheritance [Berg88, Wein 88].

Our ongoing research project is to provide minimal object-oriented extensions to a relational system to support simulation modeling. In considering extensions we heed the warning of Michael Stonebraker (INGRES), that extensions should exhibit the *spartan simplicity* of the relational model [Ston 86]. Our initial approach simply couples relational capabilities with the Modula-2 programming language in a highly intergrated fashion (e.g., no language preprocessor is necessary). In particular, OBJECTR implements relations as an abstract data type (opaque type) in the module *relation*. The operations provided in this module include the operations of relation algebra (e.g. select, project, join, union, and diff) and updating operations. In the implementation module we implement relations as a collection of tuples. Tuples are an aggregation of *atomic data types* (e.g., integers, reals, and strings). Atomic types are characterized by having an essentially fixed size and typically its internals or components are of little interest. From the point of view of users of opaque types they are atomic. Their implementation is typically structured, and sophisticated operations (procedures) can be associated with them. This is the primary extension that we provide. Note, in Section 5, we propose a bolder alternative of using an object-oriented database system coded in C++. As this research matures, we plan to compare these two alternatives (i.e., our slightly extended relational database system versus an object-oriented database system).

3.2. Query Language for OBJECTR

The query language for our simulation system provides the user interface that simplifies the processes of modeling and analysis. Because of its ease of learning and large user base, we chose the de facto standard database query language, SQL, as a starting point for our query language. Our query language provides the basic features of SQL, while adding object-oriented features by allowing procedures associated with opaque types to be executed. The basic SQL construct is a combination of SELECT, FROM, and WHERE clauses as shown below:

SELECT Attribute-List

FROM Relation-List

WHERE Predicate;

The semantics of this construct are as follows: Select (or return) values for attributes mentioned in the Attribute-List. The relations involved in answering the query must be mentioned in the Relation-List. Finally, only tuples satisfying the Predicate are returned to the user. Note, the answer to the query is itself a relation.

At this point, let us consider a simple specific example. We will present the data management support provided for the bank simulation model. In this section, we do not consider the additional complexity of information generation (i.e., we assume model instantiation (see the next section) is disabled). Simulation modelers interact with the OBJECTR system by defining a schema and by formulating queries. OBJECTR allows ordinary relational schema (no opaque types) to be specified in a Modula-2 definition module, with tuple input and output procedures specified in the corresponding implementation module. In the event that opaque types are used, the procedures that operate on the opaque type are also coded in the correponding implementation module [Mill 89]. The schema for the bank simulation model is shown below. The definition part is stored in the file bank.def. The implementation part is stored in the file bank.mod, which is generated from bank.m4, by expanding the include statement (via the UNIX m4 macro processor).

```
DEFINITION MODULE bank;

TYPE bankTup = POINTER TO bankRec;
TYPE bankRec = RECORD
    Stream:          CARDINAL;
    NumTellers:      CARDINAL;
    NumCustomers: CARDINAL;
    MeanIArrival: LONGREAL;
    MeanService:  LONGREAL;
    Throughput:   LONGREAL;
    MeanSysTime:  LONGREAL;
END (* bankRec *);

PROCEDURE DefaultParameters(t: bankTup);

PROCEDURE GetParameters(t: bankTup);

PROCEDURE WriteTuple(t: bankTup);

END bank.
```

```
IMPLEMENTATION MODULE bank;

FROM PascalIO IMPORT WRITE, WRITEc, LN;
FROM RecordIO IMPORT WriteRec;

PROCEDURE DefaultParameters(t: bankTup);
BEGIN
    WITH tˆ DO
            Stream          := 1;
            NumTellers      := 1;
            NumCustomers := 100;
            MeanIArrival := 10.0;
            MeanService  := 8.0;
    END (* WITH *);
END DefaultParameters;

PROCEDURE GetParameters(t: bankTup);
BEGIN
    DefaultParameters(t);
    WITH tˆ DO                  (* Get Parameters from SQL Query *)
            include( ‘query.par’);
    END (* WITH *);
END GetParameters;

PROCEDURE WriteTuple(t: bankTup);
BEGIN
    WRITEc(" bank.dat ", SIZE(bankRec)); WRITE(" ccclllln "); LN;
    WriteRec("ccclllln", t);
END WriteTuple;

END bank.
```

We will now consider a simple example query. Suppose a user is interested in the throughput and customer time in the system, where the mean interarrival time is 10 minutes, the mean service time is 8 minutes, and the number of tellers is 5 or less. The following query formulation will provide the desired information.

```
SELECT NumTellers, Throughput, MeanSysTime
FROM bank
WHERE (MeanIArrival = 10.0) AND (MeanService = 8.0) AND (NumTellers <= 5);
```

Query 1

Let us briefly discuss this query. The SELECT clause specifies that, for each tuple satisfying the WHERE predicate, the attributes (NumTellers, Throughput and MeanSysTime) are to be displayed to the user. The FROM clause specifies the relations involved in the query. In this case, just the bank relation (schema name = bank, model name = Bank). Finally, the predicate in the WHERE clause can be any Modula-2 Boolean expression, making it a generalization of that provided by typical implementations of SQL. As a more complex example, we could add the following attribute to the bank relation [Mill 89].

```
TimeInSystem: Statistic;        (* Opaque *)
```

Then the following query would give us more general statistical information on the time customers spend in the bank.

```
SELECT NumTellers, Throughput, DisplayStat(TimeInSystem)
FROM bank
WHERE (MeanIArrival = 10.0) AND (MeanService = 8.0) AND (NumTellers <= 5);
```

Query 2

The application of the procedure DisplayStat to the attribute TimeInSystem, which is an opaque type, is our main object-oriented extension. The complete implementation of this feature is still under development. Since users can implement as complex of procedure as they wish for abstract data types, this extension provides much power and flexibility. For example, a procedure associated with an opaque type could display a histogram.

3.3. The Information Storage/Generation Tradeoff

Generally, the relations mentioned in the Relation-List actually exist in the database (e.g., in a data file stored on disk). However, many database systems allow users to deal with relations that do not really exist. These relations are called views or virtual relations. In classical database systems, the underlying information exists to generate the information content of the view. Typically, some amount of computation (e.g., joins or calculation of averages) is required to generate this information. Furthermore, the current trend to add artificial intelligence technology to database systems, carries this idea of information generation forward. Such approaches combine basic facts with inference rules to provide powerful information or knowledge bases.

Our query driven simulation system takes this notion of a tradeoff between stored information and computation further. It allows arbitrarily complex models to generate the information. Hence, when a user queries the system and the necessary information already exists in the database, then an ordinary retrieval is used. However, if all or part of the information does not currently exist, it is generated or computed.

4. Model Instantiation

The process of model instantiation is the key aspect of our approach to automating the modeling process. Model instantiation involves extracting information from the database and the query to parameterize a simulation model(s), and then executing the model(s) to generate information that goes into the database, thus enabling the user's query to be answered.

4.1. Simple Model Instantiation

The simplest type of model instantiation occurs when only a single model is referenced and the WHERE clause is a conjunction of equality comparisons, as in Query 3. To make the process of model instantiation more concrete, let us consider for example how Query 3 is processed when the necessary information does not exist (and hence must be generated).

SELECT Throughput, MeanSysTime

FROM bank

WHERE (MeanIArrival = 4.0) AND (MeanService = 6.0) AND (NumTellers = 2);

Query 3

This query requests information on customer throughput and the mean time customers spend in the bank. When the tuples in the bank relation are examined, none will satisfy the WHERE predicate. Hence, the Bank simulation model will need to be executed to generate the information. The input parameter values needed by the model are specified by the WHERE clause. Any parameters not specified in the WHERE clause are set to their default values, which can be found in the database schema. [In this case, Stream = 1 and NumCustomers = 100.] Since this WHERE clause only has distinct values for parameters, and uses only conjunctive logical connectives, only a single model instantiation is required to provide the necessary information.

4.2. Model Instantiation with Disjunctions

A very nice feature of this simulation support system is that it makes comparisons between alternative configurations of a system easy. Suppose a bank executive wants to know whether it is better to hire one fast highly skilled teller, or two unskilled tellers which are half as fast. Assuming the cost of the first alternative is roughly equivalent to the second, the issue could be decided on the basis of which alternative has the better throughput and time in the system. The bank executive could ascertain the answer by formulating the following query:

SELECT Throughput, DisplayStat(TimeInSystem)

FROM bank

WHERE (MeanIArrival = 4.0) AND (MeanService = 6.0) AND (NumTellers = 2)

OR

(MeanIArrival = 4.0) AND (MeanService = 3.0) AND (NumTellers = 1);

Query 4

In this case, the WHERE clause is in disjunctive normal form, and to answer the query a tuple must be found in the database which satisfies one of the disjunctive terms [the Boolean expressions joined by ANDs]. If no such tuple is found, then the model is instantiated and executed twice (once for each disjunctive term).

4.3. Model Instantiation with Parameter Sets

A convenient shorthand notation for specifying families of model instantiations is provided by set notation. We process such queries by converting the set notation to disjunctive normal form. Consider, for example, Query 5.

```
SELECT Throughput, DisplayStat(TimeInSystem)
FROM bank
WHERE (MeanIArrival = 4.0) AND (MeanService = 6.0) AND
        (NumTellers IN {2..4});
```

Query 5

In this case one of the parameters is only constrained to a range of values, not just to a single value as before. In particular, we desire information on throughput and time in the system for each of the three simulation scenarios (input parameter settings). That is, we desire results for each value of NumTellers (2 to 4 inclusive). Hence, three executions of the bank simulation model will be required.

Note, at times a user may be interested in browsing through the currently available partial information, without the possibly large time penalty imposed by automatic information generation. For example, if the upper bound on the number of tellers was 50 rather than 4, a user might have to wait an intolerable amount of time for a response to his query. Hence, a switch is provided to disable automatic information generation.

4.4. Model Instantiation Algorithm

Each simulation model (e.g., Bank) has an associated relation that stores information about the scenarios (e.g., bank relation) studied with this model. The queries described above are be handled by the following algorithm (expressed as pseudocode for a UNIX shell script).

```
# Model Instantiation Algorithm
ProcessQuery < UserQuery > Response
if (-z Response) then                                # Empty file
        GenParameters < UserQuery
        foreach DisjunctiveClause
                Bank < DisjunctiveClause | UpdateDatabase
        end
        ProcessQuery < UserQuery > Response
endif
more Response
```

This shell script executes the program ProcessQuery, which takes SQL-like clauses as input, accesses the necessary relations to produce a Response file which contains all tuples matching the query. If this Response file is empty, the script will execute GenParameters to generate an input parameter file for each disjunctive term of the WHERE clause. Each parameter file contains a single tuple in which the default values have been used for any attribute not explicitly restricted by the disjunctive term of the WHERE clause. After the input parameter files have been created, the Bank simulation model is executed once for each disjunctive term, with the results being piped into the UpdateDatabase program. [A pipe feeds the output of one program into the next program as input.] The UpdateDatabase program adds the results generated by the Bank simulation program to the bank relation. Finally, after the Bank simulation model has been run for each of the disjunctive terms of the WHERE clause and the new results inserted into the database, the query is processed again. The response is displayed to the user by using the UNIX system command, more, which displays a screenful of information at a time. The actual UNIX shell script implementing this algorithm is given in the Appendix. To run, for example, Query 3, one need simply type the following:

```
rsimql bank3 bank Bank
```

where bank3 (.sql) is the query, bank (.def & .mod) is the schema, and Bank (.mod) is the simulation model.

Clearly, there is room for performance enhancements in our algorithm. However, since this is a prototype in an ongoing research project, we felt that the system should be highly modular. That this modularity has been achieved is demonstrated by the fact that the ProcessQuery program can be used in database applications unrelated to simulation modeling, and the executable model program (in this case Bank) can be executed in the ordinary fashion by a user.

4.5. Model Instantiation with Multiple Models

The most powerful type of simulation support occurs when when comparing two systems where each is represented by a different simulation model. For example, an entrepreneur may wish to know whether the time in the system for customers is greater in his bank or in his convenience store. He could formulate the following query where the join condition for the two models is that they have the same mean interarrival time.

```
SELECT MeanIArrival, DisplayStat(TimeInSystem)
FROM bank, store
WHERE (bank.MeanIArrival = store.MeanIArrival) AND
          (MeanService = 3.0) AND (NumTellers = 2) AND
          (ShoppingTime = 2.0) AND (CashierService = 1.0);
```

Query 6

Our system is not currently able to process such queries.

6. Conclusions and Future Work

Simulation support tools such as TESS (The Extended Simulation Support System) [Prit 86, Stan 87] enhance the simulationist's environment. We believe the integration of a simulation system and a database system can provide a powerful and easy to use environment for developing and analyzing simulation models. In particular, because of the complex nature of simulation data, the database system should support the storage and retrieval of objects. In our ongoing project much work remains to be done. We are currently in the process of implementing some of the more complex aspects of general model instantiation. We have also begun the substantial task of recoding SIMODULA in C++ and interfacing it with KDL. KDL itself is also under development, it currently consists of about 6000 lines of C++ code [Kess 89]. Another important area that we hope to gain experience in is developing design tools and methodologies for object-oriented databases. We are particularly interested in design methodologies for simulation support databases. This will combine elements from software engineering design methodologies, semantic data modeling, and simulation model design techniques. Figure 1 was a preliminary attempt at such a design.

8. References

[Abri 74] J. Abrial, *Data Semantics*, in **Data Base Management**, J. Klimbie and K. Koffeman (eds.), North-Holland (1974).

[Andr 87] T. Andrews and C. Harris, *Combining Language and Database Advances in an Object-Oriented Development Environment*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct 1987).

[Bank 84] J. Banks and J. Carson, **Discrete-Event System Simulation**, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1984).

[Beec 88] D. Beech, *Intensional Concepts in a an Object Database Model*, OOPSLA ’88 Conference Proceeedings, San Diego, CA (Sept. 1988).

[Berg 88] J. Bergin and S. Greenfield, *What Does Modula-2 Need to Fully Support Object-Oriented Programming?*, SIGPLAN Notices (March 1988).

[Berr 88] J. Berry, **The Waite Group’s C++ Programming**, Howard W. Sams & Company Indianapolis, IN (1988).

[Birt 73] G. Birtwistle, O. Dahl, B. Myrhaug and K. Nygaard, **Simula Begin**, Studentlitertur and Auerbach Publishers (1973).

[Bloo 87] T. Bloom, S. Zdonik, *Issues in the Design of Object-Oriented Database Programming Languages*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct 1987).

[Booc 83] G. Booch, **Software Engineering With Ada**, *The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA (1983).*

[Brun 84] G. Bruno, Using Ada for Discrete Event Simulation, Software-Practice and Experience 14, 7 (1984).

[CACI 87] CACI, **SIMSCRIPT II.5 Programming Language**, CACI, Inc., Los Angeles, Cal. (1987).

[CACI 87] CACI, **UNIX SIMSCRIPT II.5 User’s Manual**, CACI, Inc., Los Angeles, Cal. (1987).

[Capl 87] M. Caplinger, *An Information System Based on Distributed Objects*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct 1987).

[Chen 76] P. Chen, *The Entity-Relationship Model - Toward a Unified View of Data*, ACM Transactions on Database Systems (March 1976).

[Date 86] C. Date, **An Introduction to Database Systems (4th Ed.)**, Addison-Wesley Publishing Company, Inc. Reading, MA (1986).

[Duhl 88] J. Duhl and C Damon, *A Comparison of Object and Relational Databases Using the Sun Benchmark*, OOPSLA ’88 Conference Proceeedings, Orlando, FL (Oct. 1987).

[Elma 89] R. Elmasri and S. Navathe, **Fundamentals of Database Systems**, The

Benjamin/Cummings Publishing Company, Inc., Redwood City, CA (1989).

[Fish 87] D. Fishman et al., *Iris: An Object-Oriented Database Management System*, ACM Transactions on Office Information Systems (Jan. 1987).

[Grif 85] N. Griffeth and J. Miller, *Performance Modeling of Database Recovery Protocols*, Transactions on Software Engineering 11, 6 (June 1985).

[Hamm 78] M. Hammer and D. McLeod, *The Semantic Data Model: A Modelling Mechanism for Database Applications*, Proc. 1978 ACM SIGMOD International Conference on Management of Data (June 1978).

[Hugh 87] D. Hughes, *Modula-2 Co-routines and Simulation*, J. of Pascal, Ada, & Modula-2 6, 4 (1987).

[Kers 76] L. Kerschberg and J. Pacheco, *A Functional Data Base Model*, Monograph Series Technical Report, Portificia Univ. Catolica do Rio De Janeiro, Brazil, (Feb. 1976).

[Kess 89] V. Kessler and W. Potter, *New Directions for Object-Oriented Database Management Sysytems*, Proceedings of the 27th Annual ACM Southeastern Regional Conference, (1989).

[Kim 87] W. Kim, J. Banerjee, H. Chou, J. Garza and D. Woelk, *Composite Object Support in an Object-Oriented Database System*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct. 1987).

[Kim 88] W. Kim, N. Ballou, J. Banerjee, H. Chou, J. Garza and D. Woelk, *Integrating an Object-Oriented Programming System with a Database System*, OOPSLA ’88 Conference Proceeedings, San Diego, CA (Sept. 1988).

[Kowa 86] R. Kowalski and M. Sergot, *A Logic-Based Calculus of Events*, New Generation Computing 4, 1 (1986).

[Law 82] A. Law and W. Kelton, **Simulation Modeling and Analysis**, McGraw-Hill, N.Y. (1982).

[Lipp 89] S. Lippman, **C++ Primer**, Addison-Wesley Publishing Company, Inc. Reading, MA (1989).

[Maie 86] D. Maier, J. Stein, A. Otis, A. Purdy, *Developement of an Object-Oreiented DBMS*, OOPSLA ’86 Conference Proceeedings, Portland, Oregon (Sept 1986).

[Merr 87] T. Merrow and J. Laursen, *A Pragmatic System for Shared Persistent Objects*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct. 1987).

[Mill 89] J. Miller and O. Weyrich, *Query Driven Simulation Using SIMODULA*, Proceedings of the 1989 Eastern Multiconference: 22nd Annual Simulation Symposium, Tampa, FL

(March 1989).

[Mill 88] J. Miller, O. Weyrich and D. Suen, *A Software Engineering Oriented Comparison of Simulation Languages*, Proceedings of the 1988 Eastern Simulation Conferences: Tools for the Simulationists, Orlando, FL (April 1988).

[Penn 87] D. Penney and J. Stein, *Class Modification in the GemStone Object-Oriented Database System*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct. 1987).

[Pott 89] W. Potter, R. Trueblood and C. Eastman, *Hyper-Semantic Data Modeling*, Data & Knowledge Engineering (1989).

[Pott 88a] W. Potter, *KDL-ADVISOR: A Knowledge/Data Based System Written in KDL*, Proceedings of the 21st Annual Hawaii International Conference on System Science, Kailua-Kona, Hawaii (Jan. 1988).

[Pott 88b] W. Potter and R. Trueblood, *Traditional, Semantic and Hyper-Semantic Approaches to Data Modeling*, IEEE Computer (June 1988).

[Pott 87] W. Potter, R. Trueblood, C. Eastman, and M. Mathews, *KDL: A Hyper-Semantic Data Model Specification Language*, Proceeeding of the 2nd International Symposium on Methodologies for Intelligent Systems, Colloquia Program, Charlotte, NC (Oct. 1987).

[Pott 86] W. Potter and L. Kerschberg, *A Unified Approach to Modeling Knowledge and Data*, Proceedings of the IFIP TC2 Conference on Knowledge and Data (DS-2), Algarve, Portugal, (Nov. 1986).

[Prit 86] A. Pritsker, **Introduction to Simulation and SLAM II**, 3rd Ed., John Wiley & Sons, N.Y. (1986).

[Russ 83] E. Russell, **Building Simulation Models with SIMSCRIPT II.5**, CACI, Inc., Los Angeles, Cal. (1983).

[Shar 88] R. Sharma and L. Rose, *Modular Design for Simulation*, Software - Practice and Experience (October 1988).

[Ship 81] D. Shipman, *The Functional Data Model and the Data Language DAPLEX*, ACM Transactions on Database Systems (March 1981).

[Smit 77] J. Smith and D. Smith, *Database Abstractions: Aggregation and Generalization*, ACM Transactions on Database Systems (June 1977).

[Smit 87] K. Smith, S. Zdonik, *Intermedia: A Case Study of the Diffewrnces Between Relational and Object-Oriented Database Systems*, OOPSLA ’87 Conference Proceeedings, San Diego, CA (Sept. 1988).

[Stan 87] C. Standridge and A. Pritsker, **TESS: The Extended Simulation Support System**, Halsted Press, NY (1987).

[Ston 86] M. Stonebraker (Ed.), **The INGRES Papers: Anatomy of a Relational Database System**, Addison-Wesley Publishing Company, Inc. Reading, MA (1986).

[Ston 86b] M. Stonebraker and L. Rowe, *The design of POSTGRES*, ACM SIGMOD Intl. Conference on Management of Data, (1986).

[Stro 86] B. Stroustrup, **The C++ Programming Language**, Addison-Wesley Publishing Company, Inc. Reading, MA (1986).

[Sun 86] Sun, **Using Modula-2 on the Sun Workstation**, Sun Microsystems (1986).

[Sund 74] B. Sundgren, *The Infological Approach to Data Bases*, in **Data Base Management**, J. Klimbie and K. Koffeman (eds.), North-Holland (1974).

[Wegn 88] P. Wegner, *Workshop on Object-Oriented Programming ECOOP 1987*, SIGPLAN Notices (Jan 1988).

[Wein 81] D. Weinreb and D. Moon, **Lisp Machine Manual**, MIT AI Lab, (1981).

[Wein 88] R. Weiner and L Pinson, **An Introduction to Object-Oriented Programming and C++**, Addison-Wesley Publishing Company, Inc. Reading, MA (1988).

[Weyr 88] O. Weyrich, D. Brown, and J. Miller, *The Use of Simulation and Prototypes in Software Testing*, Proceedings of the 1988 Eastern Simulation Conferences: Tools for the Simulationists, Orlando, FL (April 1988).

[Wirt 85] N. Wirth, **Programming in Modula-2**, 3rd Ed., Springer-Verlag, N.Y. (1985).

7. Appendix - SIMODULA Definition Modules

```
DEFINITION MODULE simulate;
        (**************************************************************
        This module provides the basic capabilities to do process-oriented
        simulations.  It allows one to create quasi-concurrent processes which
        evolve through time by performing a sequence of activities.  The
        processes are scheduled for execution on a hidden Future Event List.
        The possible states that a process can be in are: PASSIVE - waiting
        indefinitely for some other process to activate it; ACTIVE - in the
        middle of some activity - on the Future Event List; SUPERACTIVE - the
        currently executing process; INTERRUPTED - interrupted in the middle
        of some activity; and DESTROYED - the process no longer exists only
```

```
        a pointer to it remains.
        ****************************************************************)

FROM SYSTEM IMPORT WORD;

CONST
        NOW = 0.0;

TYPE
        STATUS  = (PASSIVE, ACTIVE, SUPERACTIVE, INTERRUPTED, DESTROYED);
        ProcessPtr;  (* Opaque - pointer to process descriptor *)

PROCEDURE CREATE(VAR Process: ProcessPtr; Procedure: PROC; Attrib: WORD);
        (****************************************************************
        Create a new Process whose specification is given by the parameter
        Procedure.  Note, procedures of type PROC have no parameters.  The
        descriptor for the process will include a coroutine.  A new coroutine
        is created but control is not transferred to it.  The parameter Attrib
        is a pointer to this process's attributes.
        SIMSCRIPT II.5 EQUIVALENT:   CREATE A Process
        STATE TRANSITION:                        "" -> PASSIVE
        ****************************************************************)

PROCEDURE ACTIVATE(Process: ProcessPtr; TimeDelay: LONGREAL);
        (****************************************************************
        Activate or schedule a Process for execution.  The process is
        inserted into the Future Event List according to first its activation
        time, and second its priority.  The process's activation time is set
        to CurrentTime + TimeDelay.  To activate a process to execute
        immediately use the constant NOW.
        SIMSCRIPT II.5 EQUIVALENT:   ACTIVATE THIS Process | REACTIVATE THIS
        STATE TRANSITION:                        PASSIVE -> ACTIVE
        ****************************************************************)

PROCEDURE WORK(TimeDelay: LONGREAL);
        (****************************************************************
        Delay the current superactive process until the clock advances to
        CurrentTime + TimeDelay.  This is achieved by inserting the process
        into the Future Event List according to first its activation time,
        and second its priority, and then transferring to the process at
        the front of the Future Event List.
        SIMSCRIPT II.5 EQUIVALENT:   WORK TimeDelay UNITS | WAIT
        STATE TRANSITION:                        SUPERACTIVE -> ACTIVE
        ****************************************************************)

PROCEDURE SUSPEND;
        (****************************************************************
```

```
            Suspend (deactivate) the current superactive process.  The first
            process in the Future Event List will be the next to be executed.
            SIMSCRIPT II.5 EQUIVALENT:   SUSPEND
            STATE TRANSITION:                           SUPERACTIVE -> PASSIVE
            ******************************************************************)

PROCEDURE INTERRUPT(Process: ProcessPtr);
            (******************************************************************
            Interrupt (deactivate) a Process.  This procedure allows a process
            to interrupt some other process.
            SIMSCRIPT II.5 EQUIVALENT:   INTERRUPT THIS Process
            STATE TRANSITION:                           ACTIVE -> INTERRUPTED
            ******************************************************************)

PROCEDURE RESUME(Process: ProcessPtr);
            (******************************************************************
            Resume a previously interrupted Process.  The process will be put
            on the Future Event List with a delay equal to its remaining time.
            SIMSCRIPT II.5 EQUIVALENT:   RESUME THIS Process
            STATE TRANSITION:                           INTERRUPTED -> ACTIVE
            ******************************************************************)

PROCEDURE DESTROY;
            (******************************************************************
            Destroy or terminate the current superactive process.  The first
            process in the Future Event List will be the next to be executed.
            SIMSCRIPT II.5 EQUIVALENT:   DESTROY THIS Process
            STATE TRANSITION:                           SUPERACTIVE -> ""
            ******************************************************************)

PROCEDURE StartSimulation;
            (******************************************************************
            Start the simulation by removing the first process from the Future
            Event List and transferring to it.
            SIMSCRIPT II.5 EQUIVALENT:   Start Simulation
            ******************************************************************)

PROCEDURE ActivationTime(Process: ProcessPtr): LONGREAL;
            (******************************************************************
            Return the activation time (the time it is scheduled for execution)
            of the Process.
            SIMSCRIPT II.5 EQUIVALENT:   Time.A(Process)
            ******************************************************************)

PROCEDURE Priority(Process: ProcessPtr): CARDINAL;
            (******************************************************************
            Return the priority of the Process where 0 is the highest priority.
```

```
          SIMSCRIPT II.5 EQUIVALENT:  ?
          *****************************************************************)

PROCEDURE Status(Process: ProcessPtr): STATUS;
          (*****************************************************************
          Return the current status of the Process.
          SIMSCRIPT II.5 EQUIVALENT:  Sta.A(Process)
          *****************************************************************)

PROCEDURE Attributes(Process: ProcessPtr; VAR Attrib: WORD);
          (******************************************************************
          Return a pointer to the attributes of the Process in Attrib.
          SIMSCRIPT II.5 EQUIVALENT:  AttribName(Process)
          ******************************************************************)

PROCEDURE Time(): LONGREAL;
          (******************************************************************
          Return the current simulated time (i.e., the current value of the
          simulation clock, CurrentTime).
          SIMSCRIPT II.5 EQUIVALENT:  Time.V
          ******************************************************************)

PROCEDURE CurrentProcess(): ProcessPtr;
          (******************************************************************
          Return the process currently executing (i.e., the superactive process).
          SIMSCRIPT II.5 EQUIVALENT:  Process.V
          ******************************************************************)

PROCEDURE SetPriority(Process: ProcessPtr; ProcessPriority: CARDINAL);
          (******************************************************************
          Set the ProcessPriority for the Process to other than its default
          value.  The highest priority is 0 (the default value) while the
          lowest priority is MAX(CARDINAL).
          SIMSCRIPT II.5 EQUIVALENT:  ?
          ******************************************************************)

PROCEDURE SetWorkspace(NewSize: CARDINAL);
          (******************************************************************
          Set the coroutine WorkspaceSize to other than its default value. This
          procedure should be called before the relevant processes are created.
          SIMSCRIPT II.5 EQUIVALENT:  handled automatically
          ******************************************************************)

END simulate.
```

```
DEFINITION MODULE resources;
        (************************************************************************
        The resources module allows resources to be defined for simulation models.
        A resource consists of a number of service units (e.g., bank tellers),
        and a FCFS infinite queue feeding the service units.  A finite queue may
        be implemented by the user enforcing a constraint of the form:
                NumInQueue <= QueueCapacity
        See the simulation model "factory.mod" for a good example.
        ************************************************************************)

TYPE
        ResourcePtr;            (* Opaque data type *)

PROCEDURE CreateResource(VAR Resource: ResourcePtr; NumUnits: CARDINAL);
        (************************************************************************
        Create a resource with NumUnits service units.
        ************************************************************************)

PROCEDURE Request(Resource: ResourcePtr);
        (************************************************************************
        Request 1 unit of this Resource.  If a unit is not available, the process
        requesting service will wait in the Resource's associated FCFS wait queue.
        ************************************************************************)

PROCEDURE Relinquish(Resource: ResourcePtr);
        (************************************************************************
        Relinquish (or release) one unit of this Resource.  The first process in
        the wait queue will be ACTIVATEd, if it exists.
        ************************************************************************)

PROCEDURE NumInService(Resource: ResourcePtr): CARDINAL;
        (************************************************************************
        Return the number of units of this Resource that are busy serving
        processes.  [The rest of the units are idle and therefore available
        upon request.]
        ************************************************************************)

PROCEDURE NumInQueue(Resource: ResourcePtr): CARDINAL;
        (************************************************************************
        Return the number of processes waiting for this Resource in its queue.
        ************************************************************************)

PROCEDURE WriteResourceStat(Resource: ResourcePtr);
        (************************************************************************
        Write statistical information about this Resource. Specifically,
        information about number in service, queue length, and waiting time.
        ************************************************************************)
```

```
PROCEDURE DestroyResource(VAR Resource: ResourcePtr);
        (*************************************************************************
        Destroy this Resource.
        *************************************************************************)

END resources.



DEFINITION MODULE lists;
        (*************************************************************************
        This module provides general list handling features.
        *************************************************************************)

FROM SYSTEM IMPORT WORD;

TYPE
        ListPtr;    (* Opaque data type - Pointer to the Head of a List *)
        NodePtr;    (* Opaque data type - Pointer to a Node in a List *)

        RelationalOperator = PROCEDURE(WORD, WORD): BOOLEAN;

VAR
        DefaultLessThan: RelationalOperator;        (* Used to establish list order *)

PROCEDURE CreateList(VAR List: ListPtr);
        (*************************************************************************
        Create a List and initial it to empty.
        *************************************************************************)

PROCEDURE DestroyList(VAR List: ListPtr);
        (*************************************************************************
        Destroy the List.  Note the List must be empty.
        *************************************************************************)

PROCEDURE InsertOrder(Item: WORD; List: ListPtr);
        (*************************************************************************
        Insert Item into the ordered List.  The order is defined by the operator
        DefaultLessThan.  This operator is initialized to always return FALSE, but
        may be redefined by the user.
        *************************************************************************)

PROCEDURE InsertFirst(Item: WORD; List: ListPtr);
        (*************************************************************************
        Insert Item at the front (first position) of the List.
        *************************************************************************)
```

```
PROCEDURE InsertLast(Item: WORD; List: ListPtr);
        (**********************************************************************************
        Insert Item at the back (last position) of the List.
        **********************************************************************************)

PROCEDURE Insert(Item: WORD; Position: NodePtr; List: ListPtr);
        (**********************************************************************************
        Insert Item before the node Position in the List.
        **********************************************************************************)

PROCEDURE RemoveFirst(VAR Item: WORD; List: ListPtr): BOOLEAN;
        (**********************************************************************************
        Remove the first Item from the List.
        **********************************************************************************)

PROCEDURE RemoveLast(VAR Item: WORD; List: ListPtr): BOOLEAN;
        (**********************************************************************************
        Remove the last Item from the List.
        **********************************************************************************)

PROCEDURE Remove(VAR Item: WORD; Position: NodePtr; List: ListPtr): BOOLEAN;
        (**********************************************************************************
        Remove the Item at Position from the List.
        **********************************************************************************)

PROCEDURE RemoveAll(List: ListPtr; ItemSize: CARDINAL);
        (**********************************************************************************
        Remove all the items from the List.
        **********************************************************************************)

PROCEDURE First(VAR Item: WORD; List: ListPtr): BOOLEAN;
        (**********************************************************************************
        Retrieve the first Item in the List.
        **********************************************************************************)

PROCEDURE Last(VAR Item: WORD; List: ListPtr): BOOLEAN;
        (**********************************************************************************
        Retrieve the last Item in the List.
        **********************************************************************************)

PROCEDURE Predecessor(VAR Item: WORD; List: ListPtr): BOOLEAN;
        (**********************************************************************************
        Retrieve the previous (relative to Current) Item in the List.
        **********************************************************************************)

PROCEDURE Successor(VAR Item: WORD; List: ListPtr): BOOLEAN;
        (**********************************************************************************
```

```
          Retrieve the next (relative to Current) Item in the List.
          *************************************************************************)

PROCEDURE ListLength(List: ListPtr): CARDINAL;
          (*************************************************************************
          Return the length of the List.
          *************************************************************************)

PROCEDURE CurrentPos(List: ListPtr): NodePtr;
          (*************************************************************************
          Return a pointer to the current position (latest access) in List.
          *************************************************************************)

PROCEDURE SetCurrent(Position: NodePtr; List: ListPtr);
          (*************************************************************************
          Reset the Current pointer for List to Position.  Warning this procedure
          should used sparingly and with care since it allows users to muck with
          the internals of this module.
          *************************************************************************)

END lists.



DEFINITION MODULE statistics;
          (**********************************************************************
          This module provides facilities for gathering, storing, and displaying
          statistical summary results of simulation runs.
          **********************************************************************)

PROCEDURE Tally(VarNum: INTEGER; Val: LONGREAL);
          (**********************************************************************
          Incrementally collect statistics on sample data.
          **********************************************************************)

PROCEDURE Accumulate(VarNum: INTEGER; Val, Time: LONGREAL);
          (**********************************************************************
          Incrementally collect statistics on time persistent data.
          **********************************************************************)

PROCEDURE Min(VarNum: INTEGER): LONGREAL;
          (**********************************************************************
          Return the minimum value collected.
          **********************************************************************)

PROCEDURE Max(VarNum: INTEGER): LONGREAL;
          (**********************************************************************
```

```
          Return the maximum value collected.
          ************************************************************************)

PROCEDURE Mean(VarNum: INTEGER): LONGREAL;
          (************************************************************************
          Return the mean of the collected values.
          ************************************************************************)

PROCEDURE Variance(VarNum: INTEGER): LONGREAL;
          (************************************************************************
          Return the variance of the collected values.
          ************************************************************************)

PROCEDURE StdDev(VarNum: INTEGER): LONGREAL;
          (************************************************************************
          Return the standard deviation of the collected values.
          ************************************************************************)

PROCEDURE WriteStat(VarNum: INTEGER; VarName: ARRAY OF CHAR);
          (************************************************************************
          Write a line of summary statistics for VarNum.  Append with the string
          VarName (e.g., "Service Time").
          ************************************************************************)

PROCEDURE WriteLabels;
          (************************************************************************
          Write labels (or column headings) for the summary statistics.
          ************************************************************************)

PROCEDURE SaveVarTable(v1, v2: INTEGER; fname: ARRAY OF CHAR);
          (************************************************************************
          Save statistical information (VarTable) in file fname.
          ************************************************************************)

PROCEDURE LoadVarTable(v1, v2: INTEGER; fname: ARRAY OF CHAR);
          (************************************************************************
          Load statistical information (VarTable) from file fname.
          ************************************************************************)

PROCEDURE ResetStat(v1, v2: INTEGER; Time: LONGREAL);
          (************************************************************************
          Reset the statistical counters for VarNum's v1 to v2.
          ************************************************************************)

END statistics.
```

```
DEFINITION MODULE variates;
          (**************************************************************************
          This module provides a LCG random number generator and the random variate
          generation functions of SIMSCRIPT II.5.
          **************************************************************************)

TYPE
  STREAM = [1..10];

PROCEDURE SetSeed(i: STREAM; SeedVal: INTEGER);
          (**************************************************************************
          This procedure allows users to change the initial value (seed) for
          random number stream i.
          **************************************************************************)

PROCEDURE Random(i: STREAM): LONGREAL;
          (**************************************************************************
          This function is a LCG random number generator producing values in the
          range (0, 1).
          **************************************************************************)


(**************************************************************************
The following 9 functions generate continuous random variates.
**************************************************************************)

PROCEDURE Beta(alpha1, alpha2: LONGREAL; i: STREAM): LONGREAL;

PROCEDURE Erlang(mu: LONGREAL; k: CARDINAL; i: STREAM): LONGREAL;

PROCEDURE Exponential(mu: LONGREAL; i: STREAM): LONGREAL;

PROCEDURE Gamma(mu, alpha: LONGREAL; i: STREAM): LONGREAL;

PROCEDURE LogNormal(mu, sigma: LONGREAL; i: STREAM): LONGREAL;

PROCEDURE Normal(mu, sigma: LONGREAL; i: STREAM): LONGREAL;

PROCEDURE Triang(min, mode, max: LONGREAL; i: STREAM): LONGREAL;

PROCEDURE Uniform(min, max: LONGREAL; i: STREAM): LONGREAL;

PROCEDURE Weibull(shape, scale: LONGREAL; i: STREAM): LONGREAL;


(******************************************************************************
The following 4 functions generate discrete random variates.
```

```
*******************************************************************************)

PROCEDURE Binomial(n: CARDINAL; p: LONGREAL; i: STREAM): CARDINAL;

PROCEDURE DiscreteProb(cumprob, value: ARRAY OF LONGREAL; i: STREAM): LONGREAL;

PROCEDURE Poisson(mu: LONGREAL; i: STREAM): CARDINAL;

PROCEDURE Randi(min, max: CARDINAL; i: STREAM): CARDINAL;

END variates.
```

7.1. SIMODULA/OBJECTR Shell Scripts

```
## rsimql - run simql - simulation query language
## $1 = query, $2 = schema, $3 = model
##
rsql $1 $2 >response.tmp                # run SQL query
if (-z response.tmp) then
        echo execute model                      # empty response

        GenPAR <$1.sql                          # generate sets of parameter from query
        foreach i (*.PAR)
                mv $i query.par                 # set up include file
                m4c -g $2                       # compile model (m4 & m2c)
                m2c -M/prof/jam/simodula -M/prof/jam/objectr
                        -e $3 -lm -o $3.x       # link
                $3.x | updateRel                # execute & append tuple to database
        end

        rsql $1 $2 >response.tmp
        more response.tmp
else
        echo dont execute model
        more response.tmp
endif


## rsql - run SQL
##
cat $1.sql $2.def | /prof/jam/objectr/psql >query.mod
m2c -M/prof/jam/objectr -g query.mod -e query -o query.x
query.x
```