Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

A PROTOTYPE IMPLEMENTATION OF A MODEL MANAGEMENT SYSTEM FOR DISCRETE-EVENT SIMULATION MODELS

Melanie L. Lenard

School of Management
Boston University
Boston, Massachusetts 02215, U.S.A.

ABSTRACT

We describe a prototype implementation of a model management system (MMS) for discrete-event simulation models. The development platform is a relational Database Management System (DBMS) installed on a desktop computer.

The conceptual foundation for this MMS is Structured Modeling (SM) and Extended Structured Modeling (ESM). Using the SM and ESM frameworks makes it possible to take an integrated approach to representing and managing both models and data. It also makes possible the use of various DBMS functions to aid in the implementation of MMS functions. In effect, the MMS is constructed as a database application.

The user interface to the MMS was implemented using the Forms and Menus Facilities of the DBMS. A variety of reports documenting the model are generated using the query language of the DBMS. Most important perhaps is the code generation procedure, a complex query procedure that produces SIMSCRIPT II.5 code for simulating the model from the model representation stored in the database.

1 INTRODUCTION

Discrete-event simulation models are powerful tools for analyzing and understanding how systems work. However, building and maintaining these models is, like any other software development effort, a time-consuming, demanding, and error-prone task. Also, once models have been developed, they must be validated and verified, documented, maintained, and even reused. It is natural to think that some kind of computer-based assistance could be devised to improve the productivity of model builders and the quality of the models they produce and to support modeling activities throughout the life-cycle.

Efforts to develop software systems to support modeling efforts haven been in progress in several fields and in several contexts for at least the past 20 years. For simulation models, such efforts are called Enhanced Simulation Support Environments or Simulation Modeling Support Technologies. For a useful review article on the use of Artificial Intelligence and database technologies to support simulation modeling, see Centeno and Standridge [1992]. More generally, as an outgrowth of work on Decision Support Systems, came the realization that such systems needed model management components that could encompass many modeling paradigms (e.g., simulation models as well as optimization models, models of simultaneous differential equations, stochastic models, etc.). These systems are sometimes called model development environments (by analogy to program development environments) or model management systems (by analogy to database management systems). For an overview of these efforts see Blanning, et. al. [1992], Blanning, et. al. [1993], Shetty [1993], and Krishnan [1993].

In the work described here, we have developed a prototype Model Management System (MMS) for discrete-event simulation models. This MMS is intended to support the work of professional model builders throughout the model development life cycle, by providing its users with flexible and rapid access to models, data, and solution procedures. This prototype MMS is sufficiently general so that it could be extended to manage models from other modeling paradigms or to serve the somewhat different needs of model users.

One of the features that differentiates this MMS from other efforts in this field is that in order to achieve this general and extensible design, we chose Structured Modeling (SM) [Geoffrion (1987) and (1989)] as the conceptual foundation of the MMS. SM is a formal mathematical framework for representing a wide variety of models. SM identifies the basic elements of a model and recognizes three levels of structure for capturing the relationships among model elements. SM is in effect a ''model of models'' or meta-model.

One advantage of SM is that it provides a common representation for many kinds of models, including data models. This suggests the possibility of representing models as data. In particular, we have shown [Lenard (1986)] that by using the SM formalism, it is possible to represent at least linear programming models as a relational database.

A database representation of models can serve as the vehicle for integrating the management of models with the management of the data that support them. Furthermore, many model management functions (e.g., create, delete, update, report, query) are equivalent to database management functions [Lenard (1987) and (1988b)]. As a consequence, a database management system (DBMS) is an attractive platform for implementing an MMS. Not incidentally, various application development tools (e.g., Forms, Menus, and Report Generators) offered as part of a full-featured DBMS can accelerate the implementation process.

Recently, we have proposed [Lenard (1992)] some extensions to the original SM framework to facilitate the representation of discrete-event simulation models. In what follows, we begin with an overview of the basic concepts of this extended framework (which we call Extended Structured Modeling or ESM). Next, we describe a relational database for representing discrete-event simulation models, the design of which is based on ESM. Then, we give some details on the design and function of the prototype MMS which we have implemented using a relational DBMS. We conclude with suggestions for future development of the prototype system.

2 BASIC CONCEPTS

In this section, we will describe the basic concepts of Structured Modeling (SM) and Extended Structured Modeling (ESM) in an informal manner, using examples. (For a formal definition of SM, see Geoffrion [1989]. For a more extensive tutorial on SM, see Lenard [1988a].)

2.1 Model Elements

A structured model is composed of three basic types of elements: entities, attributes, and functions. Thus, a model comprises, not only the entities and attributes generally recognized as the components of a database, but also a new kind of component, namely functions. Entities are tangible or intangible items that are of concern to modelers and/or decision makers. There are in fact two types of entity elements: primitive entities and compound entities. Compound entities are those entities which are defined in terms of other entities (which may be either primitive or compound), while primitive entities are not defined in terms of anything else. Entities have properties associated with them, which may be specified as attributes or calculated from other elements by functions. Also, in SM logical-valued function elements are distinguished from all other kinds of functions and treated as a separate type called tests.

ESM adds three additional types of elements to those constituting SM. They are random attributes, actions, and transactions. (For formal definitions of ESM, see Lenard [1992]. For a more extensive tutorial on ESM, see the MMS Users Guide [1993].)

Random attributes are attributes that are not known in advance because they are realizations of random processes. Values for random attributes are generated according to probability distributions. These distributions, and the parameters of these distributions, are specified as part of the model.

To facilitate the use of discrete event simulation within the SM framework, ESM includes the behavior elements: actions and transactions. Action elements describe transitions. Actions specify a change to occur in the state of some model element based on the current state of that (and possibly other) elements. Actions may occur immediately after they are invoked, or after a specified delay. When the action occurs, it is assumed to occur instantaneously. In MMS, action elements have been implemented that: (a) change the value of an attribute, (b) create or delete an entity, or (c) invoke some action external to the model (e.g., cause the simulation processing to stop). Transaction elements are used to build up complex events. Transactions are sequenced lists of previously defined actions and transactions. Transactions also have a pre-condition; the transaction can be invoked only when the pre-condition is true. (Transactions are modeled after the action clusters described by Overstreet and Nance [1986].)

SM (and ESM) require that every element (except primitive entities) be defined in terms of other elements. It is further required that the system definitions be acyclic, i.e., contain no circular references. Thus every element is clearly defined and the elements it depends on can be traced back as far as one likes.

2.2 Example: Traveling Repairman Model

To illustrate these definitions, consider the classic traveling repairman model which may be described informally [Overstreet and Nance (1986)] as follows:

A single repairman services a group of identical semiautomatic machines. Each machine requires service randomly based on a time between failure which is a negative exponential random variable with parameter "mean_uptime." The repairman starts in an idle location and, when one or more machines requires service, the repairman travels to the closest failed machine. Service time for a machine follows a negative exponential distribution with parameter "mean_repair_time." After completing service for a machine, the repairman travels to the closest machine needing service or to the idle location to await the next service request. The closest machine is determined by the shortest travel time. Travel time between any two machines or between the idle location and a machine is determined by a function evaluation.

For the Traveling Repairman Model, the repairman and the machines are primitive entities; the assignment of the man to a machine would be a compound entity (that is, each assignment may be represented as a combination of one repairman with one machine.) The location of each machine is an attribute, as is its mean_uptime and its status (failed or not). The location is a "fixed" attribute (that is, a value specified in advance) while its status is a "variable" attribute (that is, a value to be determined later usually in the course of "solving" the model). The travel time between machines is a real-valued function of the attributes; testing whether or not there are any failed machines is a test defined on the attributes. The time between failures of a machine is a random attribute.

Each assignment is definitionally dependent ("calls") both the repairman and the machine to which he is assigned. The travel time associated with each assignment calls the compound entity element, which represents the assignment itself. The function element for the travel time also calls on several attributes including the location of the machine and the location of the repairman.

Changing the status of a machine from not_failed to failed is an action. To schedule a failure of a machine, one could use this action with a delay equal to the random attribute time_between_failures. Go_to_idle_location is a transaction, a set of actions which occur only if the appropriate test indicates that there are no failed machines.

2.3 Model Structure

Using SM and ESM, the elemental structure of a model is the collection of all elements -- entities, attributes, functions (including tests), random attributes, actions and transactions -- together with their definitional dependencies. In addition, SM specifies higher levels of structure (generic and modular) so that the model can be more easily comprehended and manipulated.

SM recognizes that model elements tend to occur in groups of similar elements: each group is called a genus (plural genera). All elements within a single genus must be of the same element type and they must share a "generic" calling sequence. This property, called generic similarity, means that if, say, one element of genus A calls one or more elements of genus B and genus C, then all the elements of A must call one or more elements of B and C (in the same order). A generic structure can be specified for the Traveling Repairman model by having a primitive entity genus that consists of the collection of machines, and an attribute genus that consists of the locations of these machines. Each attribute element (location) calls a primitive entity element (machine), and so the attribute genus generically calls the primitive entity genus. Since the elemental structure is acyclic, generic similarity ensures that the generic structure is also acyclic.

At a higher level of structure, groups of conceptually related genera can be grouped into modules. For example, the machines, their locations, and their status indicators, their mean_uptimes, taken together, could constitute a module. Related modules can be grouped to form even larger modules, thereby creating a hierarchy which has at the top a single module which encompasses all genera in the model. SM requires that the modular structure be such that it can be specified as an indented ordered list of modules and submodules (much like an outline of the model), where none of the calls between genera in different modules involves forward references.

Because the generic structure is acyclic, there is a default modular structure (one module for each genus and one module for the model as a whole) that will always satisfy the ordering requirement. Other more complex (and typically more useful) modular structures that better reflect the logical structure of the model can be devised. However, these must be constructed with care to satisfy the ordering requirement. The advantage of so doing is that the resulting model can be read in one pass from beginning to end and thus can be more easily comprehended.

This completes our discussion of the basic concepts of SM and ESM. Next, we will consider how an ESM representation of discrete-event simulation model may be represented in a database.

3 DATABASE DESIGN

The first step toward building a MMS using a DBMS is designing a database representation of the structured model. The database (shown in Table 1) is a somewhat revised version of an earlier proposal [Lenard (1986)]. The relations in this database describe not only the components of the model (i.e., the elements, the genera, and the modules), but also the dependencies among them, thus capturing the "structure" of the model. There are three groups of relations, namely Elemental, Generic, and Modular, corresponding to the three levels of structure in a structured model. (Dolk [1986] has developed a similar proposal in the context of a model dictionary system.)

The design of the database differs from that used by some other authors (e.g. Geoffrion [1987] and Dolk [1986]) in that all of the relations are independent of the specific model. Thus, creating a model is a matter of entering data into predefined relations rather than a matter of defining new relations. This choice of having all relations pre-defined has a two advantages: (1) since the database structure is known a priori, implementation of the MMS is easier, and (2) since all models built in this system "look alike", they are easier to understand, to combine and interchange, and to document.

Table 1: Tables (Relations) in the MMS Database

(Table names are capitalized, field names are listed in parenthesis, key fields are underlined, and descriptions of the tables are given in italics)

ACTION (modid, name, indexd, calls, rule, delay, interp) *defines and describes actions at the generic level*

AINT (modid, name, aindex, value, interp) *contains the elemental details for integer-valued attributes*

AREAL (modid, name, aindex, value, interp) *contains the elemental details for real-valued attributes*

ATTR (modid, attname, indexdby, avatype, dtype, interp) *defines and describes attributes at the generic level*

CE (modid, name, aindex, callsnam, cindex, interp) *contains elemental details for compound entities*

CMPND (modid, cename, calls1, calls2, interp) *defines and describes compound entities at the generic level*

CONTENT (modid, name, contains) *defines and describes the hierarchical relationships between modules*

E (modid, name, eindex, ename) *defines and describes entities and compound entities at the elemental level*

ENTY (modid, entyname, etype, interp) *defines and describes entities at the generic level*

ENUM (modid, constant, attname, interp) *defines and describes enumerated data at the generic level*

FNCTN (modid, funcname, indexdby, rule, calls1, calls2, dtype, interp) *defines and describes functions at the generic level*

MODELS (modid, modname, moddate, interp) *describes every model that exists in the model base*

MODULE (modid, name, interp) *defines and describes modules*

RA (modid, raname, aindex, param1, param2, param3, interp) *defines and describes the elemental details of random attributes*

RATTR (modid, raname, indexdby, distn, interp) *defines and describes random attributes at the generic level*

TRANS (modid, name, indexdby, appliesto, cond, interp) *defines and describes transactions at the generic level*

TRANSACT (modid, name, seqno, action) *defines and describes the sequence of actions constituting each transaction*

3.1 Restrictions

It should be noted that, in order to simplify the design of the database, we have taken some liberties with the original SM framework and imposed the following restrictions:

Every entity genus is self-indexed.

There is no provision for indexing variables (e.g., i, j, k).

Every function, test and attribute genus must be indexed by a single entity genus.

All function and test rules are specified at the generic level. (There are no elemental rules.)

The entity elements constituting each compound entity element must be specified at the elemental level. (There are no generic index set statements.)

No function genus may call on more than two genera (either attribute or functions). Both of the called genera must be indexed by the same entity.

No compound entity genus may call on more than two other entity genera.

There is no ordering of modules.

Elemental data (i.e., values) may be entered only for attributes of type REAL or INTEGER.

3.2 Description of Database

In the terminology of relational databases, each relation is a set of tuples consisting of a number of attributes. In what follows, however, we will refer to the relations as tables, each of which may have many records (or rows). Each row consists of a number of fields (or columns).

Each user has password-restricted access to a copy of the database. There are two sets of tables in that database: one set of tables serves as the working area and holds the data describing just one model at any given time; the other set of tables serves as the model base and holds the data for all models owned by the user. Editing of model data may be performed only on the one model contained in the working area. In the discussion that follows the structure of the model base tables will be described. See Table 1 for details.

Elemental tables are used to store the elemental data, that is, the detailed data about the entities and the values assigned to their attributes. The table **E** contains a record for each entity element, with each record consisting of fields containing: the model name, generic entity name, index, and name of the elemental entity. For example, for the generic entity ''machine'' the first (index = 1) elemental entity name might be ''old faithful''. Fixed attribute values are stored according to their data types, in two tables **AINT** (for integer-valued attributes) and **AREAL** (for real-valued attributes). The elemental details for random attributes are stored in the **RA** table. Each record in this table corresponds to one random attribute element, and includes fields for: model name, random attribute name, index, distribution parameters (3 fields), and interpretation.

Generic tables are used to store generic data. Figure 1 displays an entity-relationship diagram for the generic tables. For example, generic entities are stored in the **ENTY** table. This table contains one record for each generic entity, with fields containing: model name, generic entity name, entity type (primitive or compound), and interpretation. The field for generic entity name is used both in the generic entity table **ENTY** and in the elemental table **E**, thus linking the generic entities and their specific elemental details.

Generic attribute information is stored in the **ATTR** table, which consists of fields for: model name, attribute name, associated entity, attribute type (fixed or variable), data type, and interpretation. These tables are linked to the elemental attribute details through the attribute name. The associated entity field provides a different kind of link, by referring to the entity that a given attribute describes. Generic random attribute information is stored in the **RATTR** table. Generic function information is stored in the **FNCTN** table.

Behavior information at the generic level is stored in the **ACTION**, **TRANS**, and **TRANSACT** tables. For example, the **ACTION** table contains a record for each action, with fields for: model name, action name, associated entity, entity that is the object of the action, rule defining the action, time

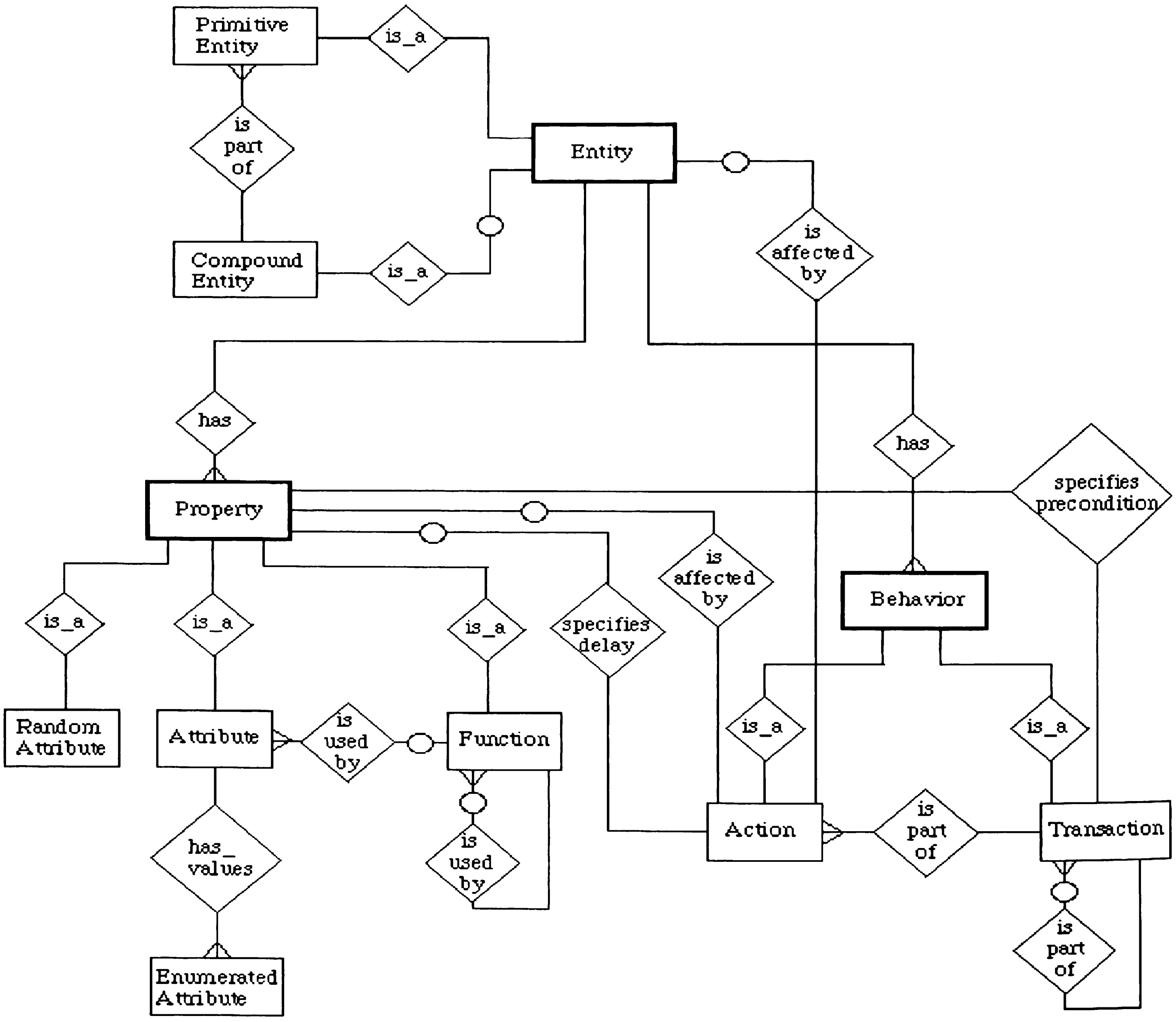


Figure 1: Entity-Relationship Diagram for MMS Generic Tables (Relations)

delay, and interpretation. The **TRANS** table contains a record for each transaction, with fields containing: model name, transaction name, associated generic entity, subset of entities that this transaction applies to, precondition, and interpretation. The information about the sequence of actions and transactions constituting each transaction is stored in the **TRANSACT** table. Each record in this table contains one action or transaction and consists of fields for: model name, transaction name, sequence number, and action name. Again, the transaction name links the actions in the **TRANSACT** table to the transactions in the **TRANS** table.

Modular relations are used to store the information representing the hierarchical modular structure within the model. The table **MODULE** contains a record for each module, with fields containing: model name, module name, and interpretation. Table **CONTENT** contains a record for each component of each module, with fields for the model name, module name and the component name.

4 THE PROTOTYPE

Using the database representation of a discrete-event simulation models described in the previous section, we have designed and implemented a prototype MMS as an application of a relational DBMS. The MMS is intended to be used by professional model builders experienced with discrete-event simulation models.

Users control the MMS through a system of menus. There are menu items that enable them to create, delete, update, and save models in the database. Also, through the menus, they can choose to access (enter, review or edit) any portion of the data representing a given model.

Users enter data describing simulation models (formulated using the concepts of ESM) into a series of forms displayed on the screen. The forms serve as the user interface to the relations (tables) in the database.

At the user's request, the MMS generates reports describing various aspects of the model at varying levels of detail. Also, on request, the MMS can (after checking the model for errors) generate and execute the code that will simulate the model, returning results to the database.

The DBMS used for this implementation is ORACLE Tools and Database for MS-DOS, Version 6. ORACLE is installed on a 386-class IBM-compatible desktop computer with 8 megabytes of RAM and 120 MB of hard disk storage and runs under MS-DOS, Version 5.0. Although the underlying model representation could, in theory, be mapped to any simulation language, this implementation generates code in SIMSCRIPT II.5.

We have designed the prototype MMS to make full use of the facilities available in the ORACLE DBMS. In the remainder of this section, we give some of the details of how this was done.

4.1 The Menu System

The menu system was created using the ORACLE tool, SQL*Menu. Every menu item invokes either a sub-menu, a form, or a procedure, as shown in Figure 2. The user moves down through the hierarchy of menus by selecting individual items on the menus; the user moves up the menu hierarchy by pressing the [Escape] key.

On entering MMS, the user sees the following Main Menu options displayed on a menu bar across the top of the screen:

Administration Entities Properties Behaviors Modules

Choosing any one of these items invokes a pop-down sub-menu with additional options. The user can exit from MMS by pressing the [Escape] key from this Main Menu.

The ``Administration'' sub-menu offers a choice of activities (e.g. create, delete, save) that can be performed on the model as a whole. The other four sub-menus offer choices allowing the user to access various portions of the data for the model in the working area as follows:

Choose Menu Item	For Access to:
Entities	Entities
Properties	Attributes, Functions, Random Attributes
Behaviors	Actions and Transactions
Modules	Modules

The organization of the menus for data access were influenced by the object-oriented paradigm. An object may be defined as an encapsulation of an entity together with its properties and behaviors. This object-orientation takes on greater importance in the design of the forms, which we describe next.

4.2 Forms

The ORACLE tool, SQL*Forms, was used to generate the forms which serve as the user interface for model manipulation and for data entry.

An example of a model manipulation form is the one which is invoked by the ``Create/select'' item on the ``Administration'' menu. This form that allows the user to enter a name and other identifying information for a new model or to select an existing model from a list of those available in the model base.

Like the menu system described above, the design of the data entry forms was influenced by the object-oriented paradigm. Therefore, in a form like the one shown in Figure 3, the top of the form displays a single entity, while the bottom of the form displays Properties (in the figure, fixed and variable attributes) of that entity. As the user scrolls through

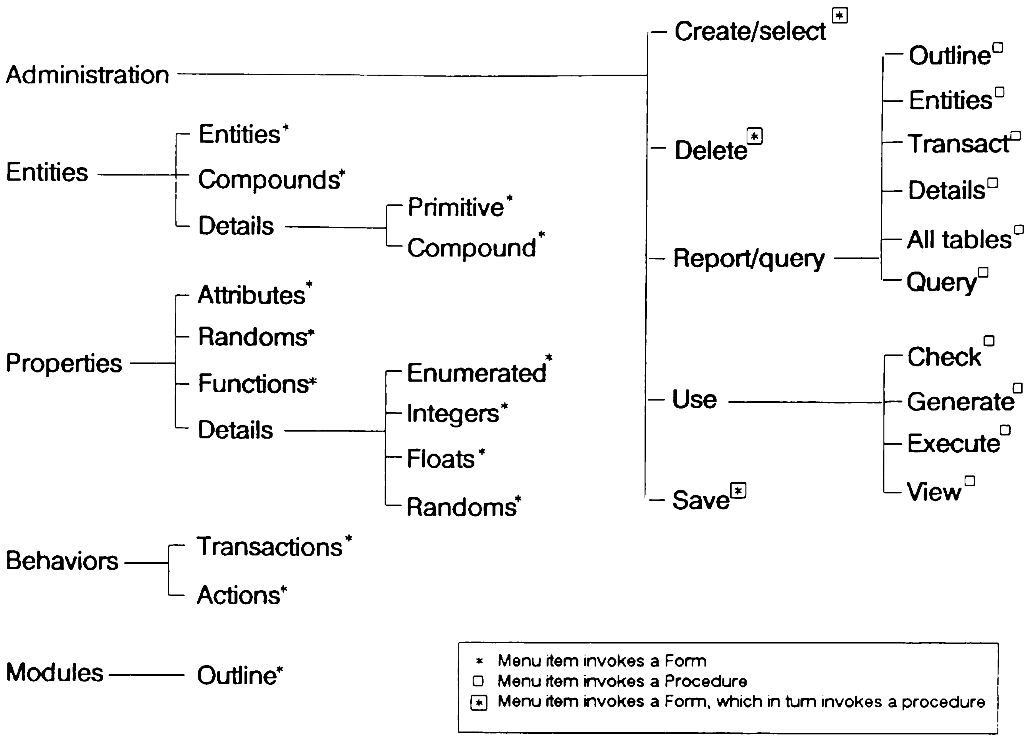


Figure 2: MMS Menu System

the entities, the corresponding attributes will be displayed in the bottom of the form. Similarly, the user is able to scroll through the list of generically defined entities when entering the elemental level detail (instances) of those entities.

A powerful feature that follows from this "object-oriented" approach is the following: when the user deletes an entity (in a form invoked from the Entities menu), all the Properties and Behaviors of that entity are also deleted as are all the elemental instances of the entity. Moreover, references to that entity (e.g., in some compound entity) are marked as unresolved.

Available operations from within a form include: scrolling through the data, creating or deleting an entire record (row), updating any field (column) in any record, and accessing lists of valid values for a particular field. Another useful operation that can be performed from a form is a query. For example, if the list of attributes for a given entity is long, the user can easily enter a query requesting display of, say, only real-valued attributes beginning with the letter "S".

Some error checking is done during data entry. For example, since attributes must be either fixed or variable, only "a" or "va" (signifying a fixed or variable attribute, respectively) is accepted as a valid entry in the "AV" (for AVATYPE) field of the form shown in Figure 3.

Changes made in data displayed in a form are transferred (committed) to the working model tables in the database when the user presses the designated function key or when exiting the form (after prompting for confirmation).

4.3 Procedures

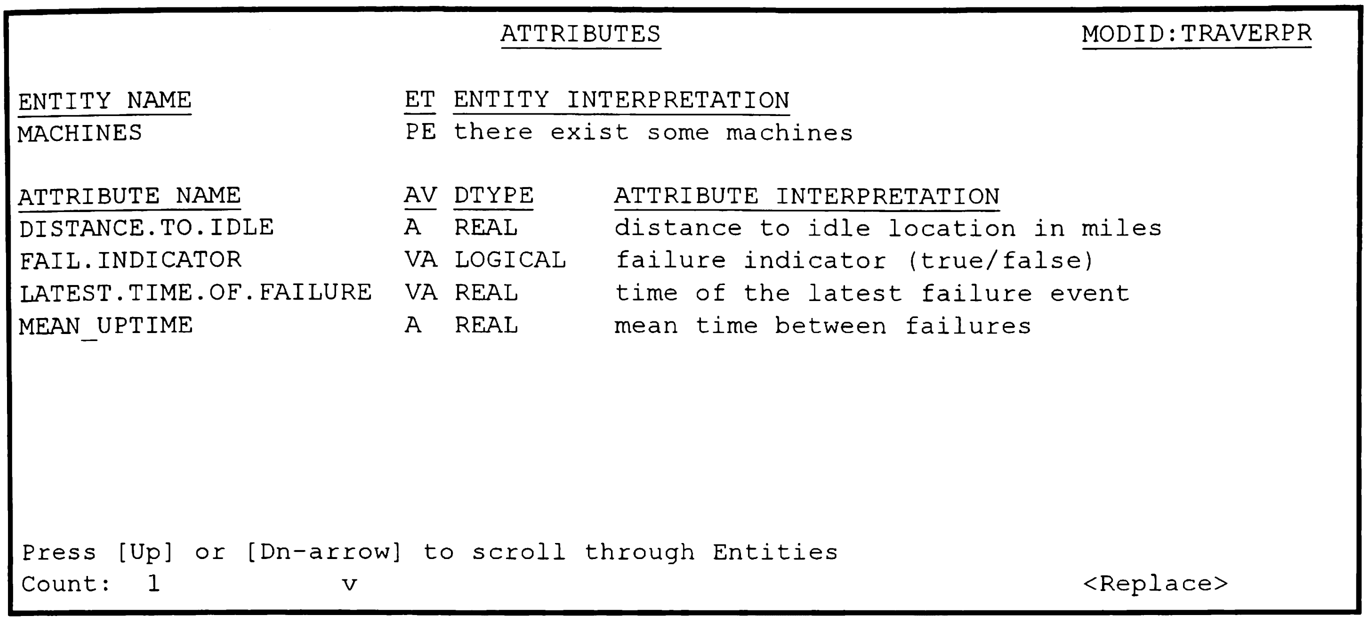
Some of the menu items and some of the forms invoke procedures to carry out their functions. These procedures are written in SQL*Plus, the ORACLE version of SQL (Standard Query Language). For example, forms for model manipulation (create, select, delete, and save) invoke procedures which make the appropriate global changes in the database (e.g., delete all records for a given model or copy all records for the working model to the model base). Also, functions like generating reports, checking for errors, and generating SIMSCRIPT II.5 code are all accomplished by procedures.

There are procedures to generate each of the following kinds of reports:

Entities -- A list of the generic entities in the model and their related properties sorted by entity type, type of property, and entity name.

Details -- A list of the generic entities and properties in the model and their related elements sorted by type, by name, by index value.

Transactions -- An outline of all the transactions and their constituent actions to a (user-specified) level of detail.



```
                         ATTRIBUTES                         MODID:TRAVERPR

ENTITY NAME              ET ENTITY INTERPRETATION
MACHINES                 PE there exist some machines

ATTRIBUTE NAME           AV DTYPE    ATTRIBUTE INTERPRETATION
DISTANCE.TO.IDLE         A  REAL     distance to idle location in miles
FAIL.INDICATOR           VA LOGICAL  failure indicator (true/false)
LATEST.TIME.OF.FAILURE   VA REAL     time of the latest failure event
MEAN_UPTIME              A  REAL     mean time between failures




Press [Up] or [Dn-arrow] to scroll through Entities
Count:  1         v                                        <Replace>
```

Figure 3: An Entity-Atribute Form

Outline -- An outline of the modular structure of the model to a (user-specified) level of detail.

All Tables -- A list of all records in all tables in the database.

There is a procedure which performs numerous integrity checks on the model, i.e., it checks for missing data and for inconsistencies within and among the modular, generic and elemental levels. For example, it checks whether a function references a non-existent attribute, or whether some elemental values are given for an attribute that is not defined at the generic level. Appropriate messages are issued if any such errors are found.

The code generation procedure is the most complex. It transforms the database representation of a model into a SIMSCRIPT II.5 program. The SIMSCRIPT II.5 program comprises a "PREAMBLE", a MAIN routine, and possibly many EVENTS and subROUTINES. The code generation procedure also generates a data file (containing the values of fixed attribute elements) which serves as input to the SIMSCRIPT II.5 program.

5 FUTURE DEVELOPMENTS

The prototype MMS described here has been used successfully to represent and simulate the Traveling Repairman model. We are currently experimenting with larger models and gradually adding features to the prototype as needed.

Also, we are currently working on an a graphical user interface. This interface will allow users to specify model elements by drawing and connecting symbols representing entities, attributes, actions, etc. The information conveyed by these symbols will be translated and input into the database. Other efforts are underway to make the system more robust, easier to use, and more complete.

Other features to be added in the near future include: (1) recording model history (when, why, and by whom created or modified), (2) more formal support for multiple versions of a model, and (3) support for less-experienced model developers, such as an on-line help system. These features are intended to support the somewhat different needs of the end-users of these models.

In the longer term, we will consider porting to more powerful workstations and perhaps using an object-oriented DBMS. Other possibilities for enhancements include: (1) adding other solution methods, such as linear programming, (2) support for model integration (i.e., combining several smaller models into one), (3) more control over the experimental design (e.g., how many replications, how much warm-up) for the simulations to be performed, and (4) developing an "expert system shell" to assist users in performing such tasks as formulating the model, choosing a solver, diagnosing errors, and explaining results.

6 CONCLUDING REMARKS

The work described here is, we believe, a significant advance in the effort to provide computer-based support for builders and users of discrete-event simulation models.

By using the Structured Modeling meta-model as a conceptual basis for the design of the system, we have laid the groundwork for an integrated approach to managing discrete-event simulation models along with models from other paradigms.

We have designed a relational database for representing structured discrete-event simulation models. This makes it possible to use a relational DBMS as the development platform. The DBMS provides the facilities for constructing the user interface and a mechanism for integrating models with data. Using the DBMS also makes available the database query language, SQL, a high-level, non-procedural language that is sufficiently versatile to describe complex tasks like code generation. Further, it puts the power of database technology (and its ongoing development) in the hands of model developers.

The implementation of this prototype MMS system, while still limited in its functionality, has served to demonstrate the feasibility of our approach to model management. Planned future enhancements will extend the range of models that can be represented and the range of operations that can be performed on them.

DEDICATION

This paper is dedicated to the memory of Joseph A. W. Smith of the U. S. Coast Guard Research and Development Center, who guided the work described here from its inception. His skill, knowledge, and vision were highly valued by all those privileged to work with him.

ACKNOWLEDGEMENTS

This research was supported in part by the U. S. Coast Guard under contract DTCG39-90-C-80829.

The author would like to thank the following members of the project team for their contributions to implementing the system: John E. Burns, Jr., Michael V. Kadyan, Daniel Z. Levy, and Michael J. Schement. Special thanks are due to Janet M. Wagner for her assistance in preparing this paper and the system documentation.

TRADEMARKS

ORACLE, SQL*Forms, SQL*Menu, and SQL*Plus are registered trademarks of Oracle Corporation. IBM is a trademark of International Business Machines Corporation. MS-DOS is a trademark of Microsoft, Inc. SIMSCRIPT II.5 is a registered trademark of CACI, Inc.

REFERENCES

Blanning, R., et. al. 1992. Model management systems. In *Information systems and decision processes*, eds. E. Stohr and B. R. Konsynski, Chapter 7. Los Alamitos, CA: IEEE Press.

Blanning, R., C. Holsapple, and A. Whinston, Eds. 1993. *Decision support systems: special issue on model management*. Amsterdam: Elsevier Science Publishers.

Centeno, and Standridge. 1992. In *Proceedings of the Winter Simulation Conference*, eds. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson, 181-189. San Francisco, CA: IEEE Press.

Dolk, D. R. 1986. *Model management and structured modeling: The role of an information resource dictionary syste.*, Dept. of Administrative Sciences, Naval Postgraduate School, Monterey, CA, August.

Geoffrion, A. M. 1987. An introduction to structured modeling. *Management Science* 33:547-588.

Geoffrion, A. M. 1989. The formal aspects of structured modeling. *Operations Research.* 37:30-51.

Krishnan, R. 1993. Model management: Survey, future research directions and a bibliography. *ORSA CSTS Newsletter.* Vol. 14.

Lenard, M. L. 1986. Representing models as data. *Journal of Management Information Systems.* 2:36-48.

Lenard, M. L. 1987. An object-oriented approach to model management. In *Proceedings: 20th Hawaii International Conference on System Sciences.* I:509-515. Los Alamitos, CA: IEEE Press.

Lenard, M. L. 1988a. Fundamentals of structured modeling. In *Mathematical models for decision support*, ed. G. Mitra. Berlin: Springer-Verlag.

Lenard, M. L. 1988b. Structured model management. In *Mathematical models for decision support*, ed. G. Mitra. Berlin: Springer-Verlag.

Lenard, M. L. 1992. Extending the structured modeling framework for discrete-event simulation. In *Proceedings: Hawaii international conference on system science*, III: 494-503. Los Alamitos, CA: IEEE Press.

Overstreet, C. M. and Nance, R. E. 1986. World view based discrete event model simplification. *Modelling and simulation methodology in the artificial intelligence era*, ed. M. S. Elzas, T. I. Oren, and B. P. Zeigler, 165-179. Amsterdam: Elsevier Science Publishers.

Shetty, B., Ed. 1993. *Annals of operations research: Special issue on model management.*

________ 1993. *Structured model management system for operational planning, installation and users guide, prototype version 2.* Brookline, MA: Crystal Decision Systems.

AUTHOR BIOGRAPHY

Melanie L. Lenard is Associate Professor of Management Science in the School of Management at Boston University. Professor Lenard received her B.S. in Chemistry from the University of Rochester and M.S. and Sc. D. degrees in Operations Research from Columbia University. Her primary research interest is in model management, including the integration of models with decision support systems. She is also a consultant on the development and management of models, and has worked in a wide range of applications in fields such as market research, health care planning, and law enforcement.