TECHNICAL ARTICLE

NEMS: A database support environment for numerical experimentation

Louis G. Birta and Maria So
Department of Computer Science
University of Ottawa
Ottawa, Ontario, Canada KiN 6N5

Simulation and computer based experimentation in general, are activities that can rapidly generate large amounts of data. The usefulness of this data depends very much on the convenience with which it can be examined, summarized and/or tabulated. In this paper, we outline the design of a generalized Numerical Experiments Management System (NEMS) that provides a framework for managing the large amount of relevant data that are typically associated with numerical studies. This includes the management of both the parameter values that are used in the experiments and the results that are produced. The work is based on a model of the problem solving environment which consists of three types of entities called solution engines, problem instances and experiments. The system environment for NEMS consists of a personal computer which runs NEMS and provides the database function, and a mainframe computer which carries out the experiments. The necessary interconnecting mechanism is assumed to be available. The organization of NEMS and its functional capabilities are described.

Introduction

In this paper we outline a software support environment that has been developed to assist in managing the data that is associated with extensive sets of numerical experiments. Such experiments could include investigations with linear and nonlinear programming codes, dynamic system studies based on the use of packages for the solution of ordinary or partial differential equations, the use of engineering design software, and simulation studies in general.

Such experimentation can be carried out from either of two perspectives; namely, that of the software developer or that of the end-user. For a software developer, the objective is generally to develop a program module that implements some algorithmic procedure (or procedures) intended for solving some specific class of problems; *e.g.* nonlinear optimization. The effectiveness of the module (or more properly, the underlying procedure) is then evaluated by testing it using numerous representative problems from the target class. On the other hand, experimentation for the end-user usually revolves around a specific problem which needs to be solved. The problem frequently involves parameters which have a range of values that must be investigated. Alternately (or perhaps simultaneously) it may be desired to explore different solution strategies based either on a variety of different models of the problem environment or on different algorithmic procedures.

The common feature in both these contexts is the need to carry out a relatively large number of numerical experiments. This naturally leads to a rapid accumulation of large amounts of data whose management in the form of printed output is a hopelessly unwieldy undertaking. The usefulness of the data depends very much on the convenience

This research was supported in part by the National Science and Engineering Research Council of Canada under Grant A8109.

with which it can be examined, summarized and/or tabulated and these tasks are clearly within the capabilities of a suitably designed data management environment. A description of the design and development of such an environment is provided in this paper.

Previous discussions in the literature about the management of numerical experimentation have occurred within the context of simulation activities. The work of Ören and Zeigler (1979), for example, suggests several file types that would be of value in a well-developed simulation environment. The Simulation Data Language (SDL) developed by Standridge (September 1981 and October 1981) incorporates a database management system specifically designed to interface with simulation software and to provide a variety of capabilities in terms of the presentation of simulation results. More recently Korn (1985) describes the DESCTOP system which integrates a simulation language environment with a file management system that maintains both program code and simulation results. The TESS system (Standridge and Pritsker 1987, Standridge and Sale 1985) is a relatively new and particularly comprehensive development. This high level language interface supports a wide range of the requirements of the simulation life-cycle for studies carried out within a particular programming environment.

The work described in this paper shares a degree of common functionality with these various related developments. A principal difference, however, relates to the scope of applicability. As indicated above, previous work has focused exclusively on data management issues within a simulation context. On the other hand, the present work is a general purpose tool which is applicable to numerical experimentation in an arbitrary context. It provides a general management function both for the setting up of experiments and for the data that is associated with the experiments (experimental conditions and output). Also, each of these related developments has evolved around a specific simulation language environment. TESS, for example, is intimately linked to SLAM II. In contrast, the management tool outlined in this presentation is not biased toward any programming language. In other words, the management function which it provides is independent both of the specific nature of the experiments being carried out and of the language used to create the modules that participate in these experiments, (e.g., the "models" and the model processing software).

Conceptual Framework

The Numerical Experiments Management System (NEMS) manages three classes of objects; namely solution engines (SE's), problem instances (PI's) and experiments. At any particular moment, the NEMS database will generally be populated with numerous occurrences of each of these object types.

An SE is unit of program code (or simply a "program") that has been designed and written to carry out some specific problem solving task; *i.e.*, to solve problems of a particular class. Representative examples of SE's are:

(a) a linear programming package which generates a solution to linear programming problems,

(b) a program package which contains a collection of procedures for solving ordinary differential equations (some of the procedures in the package may be better suited to solving certain types of problems; *e.g.* stiff equations)

(c) a simulation package specifically designed to accommodate performance models of computer systems.

An SE is characterized by two groups of attributes called defining attributes and operational attributes respectively. For each SE known to it, the NEMS database maintains values for the attributes in both these categories. Defining attributes are primarily concerned with characterizing the underlying structure of the SE and once specified, they remain invariant over the life of the SE.

Typically, a solution engine is designed to function over a range of circumstances; *e.g.*, various sizes of some input data array, various values of some termination condition or various levels of solution accuracy. We refer to attributes of this type as "parameters". A distinguishing feature of a parameter is that it must be assigned a value before the engine can be executed. After the execution of an SE, values are acquired by a designated set of internal variables. The problem solution that is sought is, at least in part, contained among these values. These special variables whose values characterize the solution generated by the SE are referred to as output variables or simply "variables". The set of parameters and variables associated with a solution engine constitute its operational attributes. Typically one or more of these changes in value from one use of the solution engine to another.

As suggested above, an SE is a computational work-horse designed to solve problems of some particular class. Its activation is meaningful only after it has been coupled in some appropriate way with a specific problem of the target class. The nature of the problems in the target class and their presentation format are intimately linked to the SE under consideration. If, for example, the SE in question is the SIMSCRIPT processor, then a problem from the associated target class will exist as a SIMSCRIPT program that has been written to conform to the syntactic requirements imposed by the SIMSCRIPT language processor. Furthermore, this program will likely be a representation of some simulation problem that is of interest.

Each specific problem within the target class of some SE is called a problem instance (PI) and is assumed to exist as a unit of program code. Like a solution engine, a problem instance is characterized by defining attributes and operational attributes. The nature and role of these attributes parallel the equivalent concepts applied to SE's. Each problem instance is regarded as being associated with a unique solution engine; *i.e.*, a PI can be "solved" only by one particular SE. The associated SE is referred to as the PI's master and correspondingly, one of the defining attributes of each PI identifies this particular SE. In effect, an SE corresponds to a problem solving tool, while a PI corresponds to a problem occurrence that is compatible with one of the SE's known to NEMS.

The notions of an SE and a PI as outlined above, are very general and, in fact, admit situations that are outside the intended scope of the NEMS concept. For example, it would be possible to identify a payroll program (system) as an SE, in which case the associated PI would be an employee file containing employment data for some particular week. However this example is not entirely consistent with

certain implicit assumptions inasmuch as there is no structural feature in the PI that could be regarded as invariant; *i.e.* the PI is "all data". More specifically, our interpretation of a PI always incorporates a loose notion of a "model".

Taken in isolation, neither an SE nor a PI is an executable unit. Rather it is the union of some SE with some PI within its domain which gives rise to an executable program module and the execution of such a module corresponds to an "experiment". Typically the outcome of the experiment is regarded as a "solution" to the problem represented by the PI.

Prior to carrying out an experiment, values need to be prescribed for the parameters of both the SE and the PI, and as a consequence of the experiment, each output variable of both the SE and the PI acquires a value. The fundamental task of NEMS is to capture the values of the parameters and variables associated with an experiment and to organize this data in a meaningful (*i.e.* useful) way for the user.

The NEMS Operational Environment

The typical NEMS user is concerned with multiple SE's and associated PI's and experiments. The fundamental task of NEMS is to provide a database support environment for manipulating these objects in terms of the values associated with their various attributes. This support environment has been designed around the relational database model, because of the relative ease with which the objects in question can be mapped onto the tabular structures upon which the relational model is based.

Relationally oriented database management software is widely available for personal computers (pc's) and the nature of the database management requirements within NEMS is entirely consistent with the capabilities of such systems. Consequently the approach taken in the development of NEMS was to create around one such system a superstructure designed to accommodate the specialized needs of the experimentation activity. The specific system chosen to provide this underlying database management function was dBASE III.

The pc/dBASE III environment supports only one aspect of the NEMS concept; namely the data management activity itself. However the NEMS user's primary (productive) activity is with carrying out numerical experiments and therefore the environment for carrying out this activity must also be considered. Two alternatives are clearly available; namely, the pc itself or a remote mainframe. In many circumstances, the program code for the SE's to be supported by NEMS may be extensive and execution during the experimentation phase may require substantial cpu power in order to achieve reasonable response time. Such features could make a wholly pc-based environment impractical. To avoid such limitations, the current NEMS implementation is based on a dual environment in which a pc provides the data management function (using dBASE III) and a mainframe provides the computing power for the experimentation. The program code for the various SE's and PI's that are known to NEMS is assumed to reside in appropriate files on the mainframe. In addition, the necessary communication link between the pc and the mainframe is assumed to exist.

Parameters and Variables and their Characteristics

The parameters and variables within an SE or PI constitute its operational attributes. A prerequisite to the initiation of an experiment (between a specific SE and PI) is the specification of values for all parameters that exist within the SE/PI combination. A consequence of the experiment is the acquisition of values by all variables that exist within the SE/PI combination.

Value assignment to parameters takes place within a framework that is driven by a set of attributes associated with each parameter (the specification of values for these attributes takes place during the Create phase, as outlined in section 6). In particular each parameter within either an SE or PI has a type (numeric or character string) and a default value. Numeric types have, in addition, a mode (integer or real), and an upper-bound and a lower-bound value. A parameter of numeric type may have a scalar or a vector structure while only the former option is available for parameters whose type is character string. When a parameter has a vector structure, additional attributes that relate to its length, become relevant. A vector parameter has a maximum length and an active length where the latter is the actual length of the vector during any particular usage of the module to which it belongs.

Frequently there exist parameters which an SE shares with every PI that it owns. Consider for example, a solution engine designed to carry out a nonlinear function minimization task. When applied to any particular problem, a fundamental value that is likely required by the solution engine is the size of the problem; *i.e.*, the number of free parameters that are available for adjustment during the minimization activity. This particular value is clearly a characteristic feature of each problem. In other words the value of this parameter originates within the problem specification but is none the less required by the associated SE.

Like all parameters, such a shared parameter has attributes. These could be declared during the creation of each problem instance. However such an approach is repetitive and error prone because care must be taken to ensure a consistent attribute specification. A far more convenient approach is to specify the attributes of the shared parameter during the creation of the associated SE. In this way the specification takes place only once and thus the consistency issue is avoided. It is this latter approach that is embedded in NEMS and its implementation requires that appropriate steps be taken to ensure that these specifications are automatically "reflected into" (inherited by) every PI that is owned by the SE. Values assigned to such reflected (shared) parameters are stored within the storage structures of the PI's.

The reflected parameter concept is an example of the concern in NEMS with simplifying/minimizing user input, thereby reducing the likelihood of erroneous or inconsistent data. Another example of this feature is the flexibility that is provided for specifying how the active length of a vector-valued parameter is to be established. The following four options are available:

(i) it may be fixed (and hence invariant over all usages of the associated module)

(ii) it may be explicitly assigned a value by the user immediately prior to the initiation of the experiment involving the module

(iii) it may acquire its value indirectly from some scalar parameter associated with the experiment

(iv) it may inherit the same active length as that of some other vector parameter (or variable) associated with the experiment.

One of the attributes relevant to a vector-valued parameter relates to this option. Notice that only the second option gives rise to an explicit input from the user at execution time.

Variables are the second type of operational attribute of both SE's and PI's. Variables, like parameters, acquire values. The fundamental difference between variables and parameters is that variables acquire their values as a consequence of an experiment; the experimenter has no explicit control over these values, as is the case with the values of parameters. Variables, like parameters, have a type (numeric or character string), and variables of numeric type have a mode (integer or real) and a structure (scalar or vector). The notions of default value, and upper and lower bound value do not have relevance to variables. Variables however, have a distinctive structural option called a cluster-list (c-list). A cluster-list can be regarded as an ordered set of vectors. An application for this concept would be in characterizing the solution trajectory of a set of ordinary differential equations. Suppose, for example, that a problem of this class contains a set of n first order differential equations. From the problem solver's perspective, the solution will likely consist of the set of vectors:

$$(t_k, x_1(t_k), x_2(t_k), \ldots\ldots x_n(t_k); k=1, 2, \ldots\ldots m$$

The first entry, t_k, in each such vector is a value of the problem independent variable and the remaining entries are the solution values at t_k. This set of vectors is regarded as a c-list within NEMS. Each c-list has a size which is the number of entries in each component vector (n+1 in the above example). Note however that the number of vectors comprising any particular c-list is an experiment dependent value, similar to the value of a variable.

The essential information relating to an experiment is represented by the set of corresponding parameter/variable values (it could be argued that the experiment is this set of corresponding values!). The data management task is fundamentally the task of capturing each such set of corresponding values, making them conveniently accessible to the experimenter and ensuring that their interrelationship is maintained.

The NEMS Storage Structures

The underlying NEMS database is relational and hence consists of a collection of inter-related tables. These tables are transparent to the user but the system response to input obtained from user dialogs carried out during the NEMS modes (see following section) , has an impact on the existence of these tables and on the values stored within them. A global view of the tables and their hierarchically structured inter-relationships is shown in Figure 1. The purpose of each of the NEMS tables is summarized in Table 1. Additional details are given in the following discussion.

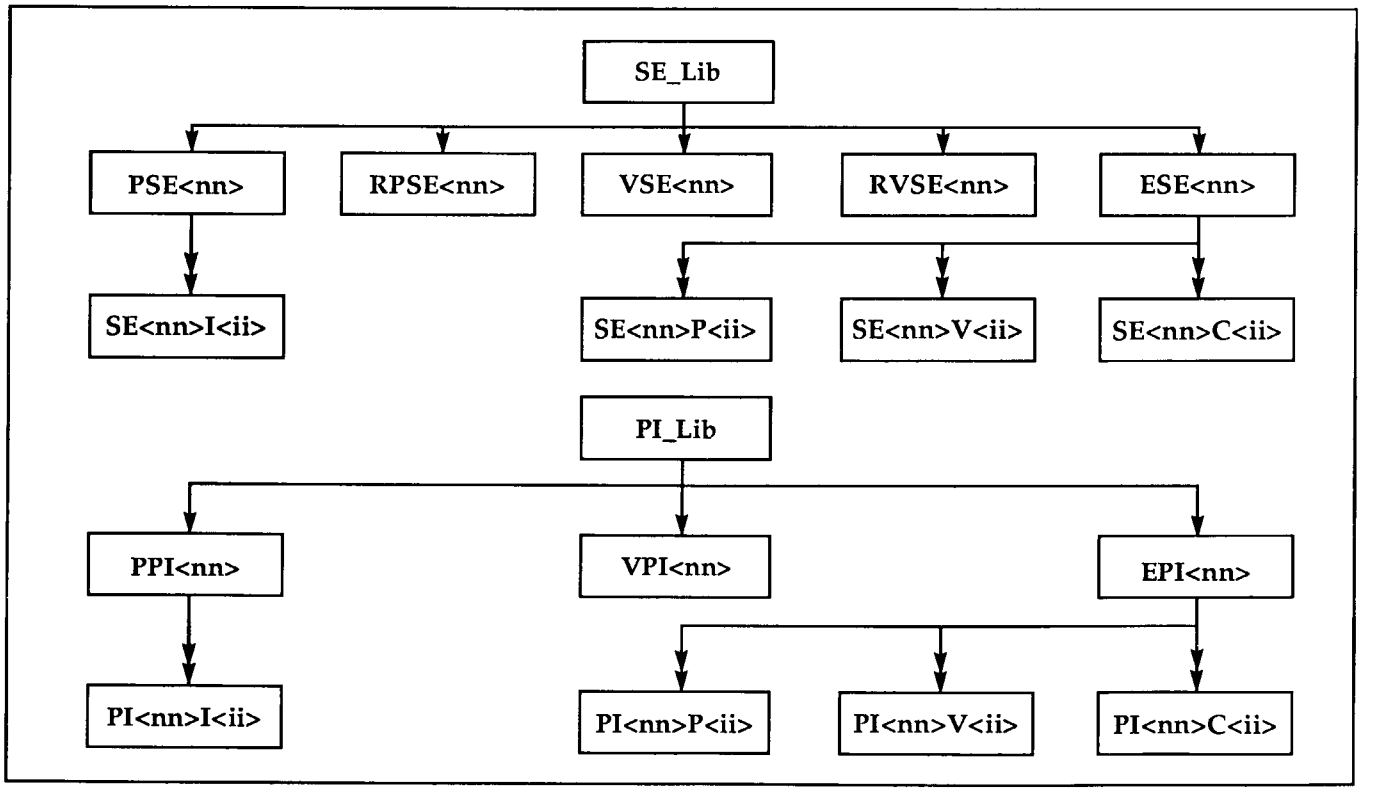

Figure 1. Interrelationships among the NEMS Relations

Table 1. General Specifications for the NEMS Relations

RELATION NAME	SPECIFICATION
SE_Lib	Maintains a directory of all SE's known to NEMS, characterized in terms of values for their defining attributes.
PSE<nn>	Stores values for the atttributes of each non-reflected parameter belonging to SE<nn>; *i.e.* belonging to the SE whose internal identifier is SE<nn>
SE<nn>I<ii>	Stores the default values and the upper and lower bound values for each component of the non-reflected vector parameter within SE<nn> whose internal identifier is P<ii>.
RPSE<nn>	Stores the values for the attributes ofeach reflected parameter belonging to SE<nn>.
VSE<nn>	Stores values for the attributes of each non-reflected variable belonging to SE<nn>.
RVSE<nn>	Stores values for the attributes of each reflected variable belonging to SE<nn>.
ESE<nn>	Stores the data associated with each of the experiments carried out with SE<nn>; *e.g.*, the associated PI used in the experiment, the values of scalar non-reflected parameters, and the values of scalar non-reflected variables.
SE<nn>P<ii>	Stores the value acquired during each experiment with SE<nn>, by the non-reflected vector parameter internally identi fied as P<ii> within SE<nn>.
SE<nn>V<ii>	Stores the value acquired during each experiment with SE<nn>, by the non-reflected vector variable internally identified as V<ii> within SE<nn>.
SE<nn>C<ii>	Stores files identification information, for each experiment with SE<nn> relating to the C-list variable internally identified as V<ii> within SE<nn>.
PI_Lib	Maintains a directory of all PI's known to NEMS, characterized in terms of values of their defining attributes.
PPI<nn>	Stores values for the attributes of every parameters belonging to PI<nn>; i.e. belonging to the PI whose internal identifier is PI<nn>.
PI<nn>I<ii>	Stores the default values and the upper and lower bound values for each component of the vector parameter within PI<nn> whose internal identifier is P<ii>.
VPI<nn>	Stores values for the attributes of every variable belonging to PI<nn>.
EPI<nn>	Stores the data associated with each of the experiments carried out with PI<nn>; e.g., values of scalar parameters and variables.
PI<nn>P<ii>	Stores the value acquired during each experiment with PI<nn>, by the vector parameter internally identified as P<ii> within PI<nn>.
PI<nn>V<ii>	Stores the value acquired during each experiment with PI<nn>, by the vector variable internally identified as V<ii> within PI<nn>.
PI<nn>C<ii>	Stores file identification information, for each experiment with PI<nn>, relating to the c-list variable identified as V<ii> within PI<nn>.
EXP_Lib	Stores a characterizing tuple for every experiment carried out with NEMS; the information includes the names of the SE and PI used in the experiment.

In Figure 1, the tables are represented by rectangles whose designation follows a particular naming convention. This convention has its basis in the internal designations that NEMS assigns to all SE's and PI's that are known to it and to all parameters and variables associated with these SE's and PI's. Under this convention, each SE (PI) is assigned a sequential internal identifier of the form SE <nn> (PI <nn>) where <nn> represents a two digit integer. Similarly each parameter (variable) within a module has an internal identifier of the form P<nn> (V<nn>). A link between two tables in Figure 1 implies that each tuple (*i.e.*, row) in the superior table may have associated with it zero or one table of the connected type. The root table, SE-Lib, maintains the directory of all SE's known to NEMS. The column headings (attributes) of this relation correspond to the SE defining attributes as outlined in the section on The Create Mode.

The various non-reflected parameters associated with the SE identified internally as SE<nn> are characterized within the relation PSE<nn>. This characterization consists of specifying values for the attributes of each parameter belonging to SE<nn>; e.g. its name, type, mode, structure and where appropriate, its default value, upper bound and lower bound values, maximum size and means for determining active size (section 3). This characterizing information is inserted into prescribed column locations and each such parameter gives rise to a distinct new row in PSE<nn>. Special attention must, however, be given to the case where a parameter has a vector structure because the collection of associated default and upper and lower bound values cannot be accommodated in PSE<nn>. In such circumstances (i.e. when a parameter P<ii> of SE<nn> is vector valued) a secondary table, called SE<nn> I<ii>, is created to provide the storage area for this data.

The characterization of each reflected parameter associated with SE<nn> takes place in the relation called RPSE<nn>. The attributes (column headings) of this relation are a subset of those of PSE<nn> because reflected parameters have fewer properties within the context of their existence within an SE. In particular, the specification of default and lower and upper bound values is deferred to each PI context and consequently RPSE<nn> does not give rise to a dependent relation equivalent to SE<nn>I<ii>.

The relation VSE<nn> provides the storage structure for characterizing each of the non-reflected variables associated with the SE whose internal identifier is SE<nn>. Each such variable gives rise to a distinct tuple (row) within this relation and the attributes of each tuple (*i.e.* the column headings of VSE<nn>) include the variable's name, type, mode and structure. In addition, if the variable has a vector or c-list structure, then the tuple components also include maximum size and information that indicates how the active size is to be established.

The specifications for each of the reflected variables associated with SE<nn> are maintained in the relation RVSE<nn>. The structural organization of this relation parallels that of VSE<nn>.

The purpose of the relations PSE<nn>, RPSE<nn>, VSE<nn> and RVSE<nn> is to maintain information that suitably characterizes the various parameters and variables associated with the SE that is internally identified as SE<nn>. Each use of this SE in an experiment is associated with a set of values for these parameters and variables. The

relation ESE<nn> provides the storage structure for maintaining this value information.

Each row in ESE<nn> corresponds to an experiment with SE<nn>. Each such experiment is identified with a unique (sequential) identification number and must be associated with some (compatible) PI. This information is contained among the tuple components of each entry in ESE<nn>. The remaining tuple components are the values of the non-reflected parameters and variables that are associated with the experiment, (the values of reflected parameters and variables are maintained in storage structures associated with the problem instance used in the experiment). When a parameter (*e.g.* P<ii>) or a variable (*e.g.* V<ii>) is vector valued, then the value data is stored in the auxiliary relations SE<nn> P<ii> and SE<nn>V<ii> respectively.

The value data for c-list variables is handled in a distinctive way, because an underlying assumption is that this data can become extensive and may burden the possibly limited storage facilities of the pc. As a consequence, the convention in NEMS is to leave c-list value data in a file on the mainframe which executes the experiment. Associated with each c-list variable V<ii> within SE<nn>, there is an auxiliary relation called SE<nn>C<ii>. Each tuple in this relation contains the information necessary to access the value data for V<ii> for some particular experiment with SE<nn>; *e.g.* the name of the file that contains the data.

An analogous collection of hierarchically organized relations exists for maintaining the relevant information about the PI's known to NEMS. The root relation in this hierarchy is PI-Lib and each tuple in this relation includes the values of the defining attributes for some PI (see the section on The Create Mode). It can be noted in Figure 1 that the PI hierarchy does not include relations equivalent to RPSE<nn> and RVSE<nn>. This asymmetry arises because the notion of reflected parameters and variables is distinctive to SE's. Parameters and variables reflected into a PI are not distinguishable from "normal" parameters and variables belonging to the PI and hence are accommodated in the relations PPI<nn> and VPI<nn> respectively.

One additional relation exists which is not shown in Figure 1. This is the relation EXP-Lib which acquires a new tuple each time an experiment is carried out under NEMS. The tuple components of EXP-Lib include an experiment identifier (sequentially assigned), the date of the experiment and the internal identifiers of the SE and PI used in the experiments.

The NEMS Modes

NEMS has four operational modes which are referred to as the create, execute, query and delete modes. The following discussion outlines the principal features of each of these modes.

The Create Mode

NEMS has been designed to accommodate an arbitrary number of both solution engines and problem instances. The create mode enables the user to introduce into his particular NEMS environment the existence of such entities. (It should be appreciated here that the process of creating the existence within NEMS of an SE or PI entity is distinct from the activity of creating the actual program code for that entity). The create activity involves the specification of values for the set of defining attributes for the module (SE or PI) and the specification of values for the attributes of its parameters and variables. This activity, which is carried out via a dialog with the user, results in the insertion of entries in existing storage structures (tables/relations) as well as the creation of new storage structures.

The defining attributes for a solution engine are as follows:

(a) user assigned name for the SE

(b) internal identifier for the SE

(c) creation date

(d) name of the "owner" of the SE

(e) number of parameters (exclusive of reflected parameters)

(f) number of reflected parameters

(g) number of variables (exclusive of reflected variables)

(h) number of reflected variables

(i) the name of the command file that activates the SE

(j) the internal arithmetic precision used by the SE (single or double)

(k) number of usages to date of the SE

With regard to item (i), recall that the SE is assumed to exist as a (compiled) program module whose execution must be activated; the directives in the named command file control this execution.

The defining attributes for a problem instance are similar to those of a solution engine; namely,

(a) user assigned name for the PI

(b) name of the "SE master" for the PI

(c) number of parameters (exclusive of reflected parameters)

(d) number of variables (exclusive of reflected variables)

(e) the internal arithmetic precision of the PI

(f) the name of the "owner" of the PI

Following the specification of values for the defining attributes of the module being created (either an SE or PI), the NEMS dialog proceeds to obtain from the user the values for the basic attributes of each of its parameters and variables. The basic attributes for all parameters and variables of both SE's and PI's consist of the name of the parameter/variable and its type, mode and structure. The activity following the specification of values for the basic attributes depends on whether an SE or PI is being created and on whether the identifier in question is a parameter or a variable.

The following rules apply for parameters within an SE:

(a) If the parameter is scalar and reflected, then values are required only for its basic attributes.

(b) If a parameter is scalar but not reflected, then a default value is required. If, in addition, the parameter is of numeric type then upper and lower bound values are also required.

(c) If a parameter is vector-valued (hence numeric) and reflected then a maximum length is required together with the means for establishing its active length, (this

could involve the specification of an identifier from which the active length is acquired).

(d) If a parameter is vector-valued but not reflected then a default value and upper and lower bound values for each of its components are required. In addition a maximum length value is required together with the active length specification method (as in (c) above).

It can be noted that the principal difference between the reflected and non-reflected cases relates to the acquisition of default and upper and lower bound values. In the former case these values are not assigned at SE creation time because these parameters have no explicit existence within the SE (their existence materializes within each of the PI instances that belong to the SE).

Since variables do not have default or upper and lower bound attributes, there is no distinction between the treatment of reflected and non-reflected variables. During the creation phase, therefore, the processing of each variable simply requires the user's specification of the variables name, type, mode and structure. In addition, however, for variables with vector or c-list structure, the maximum size value is required together with the mechanism to be used for establishing active size at execution time.

The creation of an SE or a PI results in two types of change to the NEMS database. The more straightforward of these is the insertion of a new tuple in either the relation SE-Lib or PI-Lib (as appropriate; see the section on The NEMS Storage Structures) to register the existence of the new module. The second type of change corresponds to the creation of various new relations which will maintain the values of the parameters and variables of the module which are associated with its usage in experiments.

The Execute Mode

The purpose of the execute mode is to provide the framework for carrying out an experiment. The activity of the execute mode is organized into three phases which correspond to a preamble (experiment set-up), experiment execution and result retrieval. The preamble is driven by a dialog with the user to establish the particulars of the experiment to be carried out. The purpose of this dialog is to:

(i) identify the specific solution engine and problem instance that are to be used in the experiment

(ii) assign values to the parameters within the designated SE and PI , and

(iii) specify, if required, the active lengths of vector valued parameters or variables, or of c-list variables, within the designed SE and PI.

The names of the SE and PI to be used in the experiment, provide the data which characterizes (in part) a new instance of an experiment object and this, in turn, gives rise to a new entry in the relation EXP_Lib. If the designated SE and PI have the internal identifiers SE<nn> and PI<nn> respectively, then the parameter characterization information that is maintained in the relations PSE<nn> and SE<mm>I<ii> (for the solution engine) and in the relations PPI<mm> and PI<mm>I<ii> (for the problem instance) is used to guide the user dialog for obtaining parameter values. During this dialog, the user is given the option of either assigning the predefined default value for each parameter or explicitly specifying a value. Furthermore, in the case of a parameter of numeric type, any user supplied value is checked against the predefined mode and the stored upper and lower bound values. Deviations are intercepted and corrections are requested from the user. The parameter value data itself is stored as part of newly created tuples in the relations ESE<nn> (for the solution engine) and in the relation EPI<mm> (for the problem instance).

The principal activity in the preamble is the assignment of values to the various parameters relating to the experiment. However, as indicated previously, there is another possible type of data requirement; namely the active length values for parameters or variables having vector structure or the active size values for c-list variables. A requirement of this type is detected from the specifications contained within the PSE<nn> and PPI<mm> relations and, when necessary, the required length/size data is requested from the user.

The order in which parameter values are obtained from the user during the preamble is not entirely arbitrary because the active length/size of a vector/c-list can be defined in terms of other parameters and variables. Furthermore the parameter or variable from which inheritance takes place may exist within the "other" module; *e.g.*, the active size of a vector variable within an SE may be implicitly defined in terms of a scalar parameter within the PI being used in the experiment. An appropriate sequencing procedure is embedded within NEMS to ensure that such interdependencies between data values are properly accommodated during the user dialog of the preamble.

The steps within the second and third phases of the execute mode are conceptually straightforward. The second phase begins after all necessary parameter data has been acquired and the first step is the automatic creation of a file of predefined format which includes the parameter values stored in the appropriate tuples of the relations ESE<nn> and EPI<mm>. With the current version of NEMS, the user now has the responsibility of (a) transferring the parameter value file to the remote mainframe where the code execution for the SE/PI combination will actually take place and (b) activating an existing command file on the mainframe to execute the designated SE (an embedded accessory activity here is the reading of the parameter value file). Step (b) then corresponds to carrying out the specified experiment. As a result of appropriately placed output commands within the SE/PI code, a file containing values for the variables associated with the experiment execution, is created.

In the third (final) phase of the execute mode, the output file is transferred back to the pc (in the present version of NEMS, this is again the responsibility of the user). Upon completion of this file transfer, the NEMS monitor automatically transfers the data values into the appropriate tuples of the relations ESE<nn>, SE<nn>P<ii>, SE<nn>V<ii> (for variable values relating to the solution engine) and/or into the appropriate tuples of the relations EPI<mm>, PI<mm>P<ii>, PI<mm>C<ii> (for variable values relating to the problem instance). At this point the experiments execution is completed and control returns to the main (i.e., highest level) NEMS menu.

The Query Mode

Large amounts of experimental data will typically be accumulated in the NEMS database relating to many solution engines and problem instances. Since NEMS is built upon dBase III, the user naturally has the option of accessing the dBase III tables directly through the underlying retrieval facilities. This route will generally not be attractive to most users since it requires familiarity with both the internal structure of the tables and the dBase III retrieval procedures.

As an alternative, NEMS includes a menu driven query subsystem for browsing through the accumulated data. This subsystem is tailored to the typical needs of the NEMS user and is organized around three basic menus called the SHOW/DESIGNATE menu, the REFINE menu and the DISPLAY menu. The structure of the query subsystem and the options provided by the three menus are shown in Figure 2 (the basic menus are represented by the nodes labelled 1, 2 and 3 respectively).

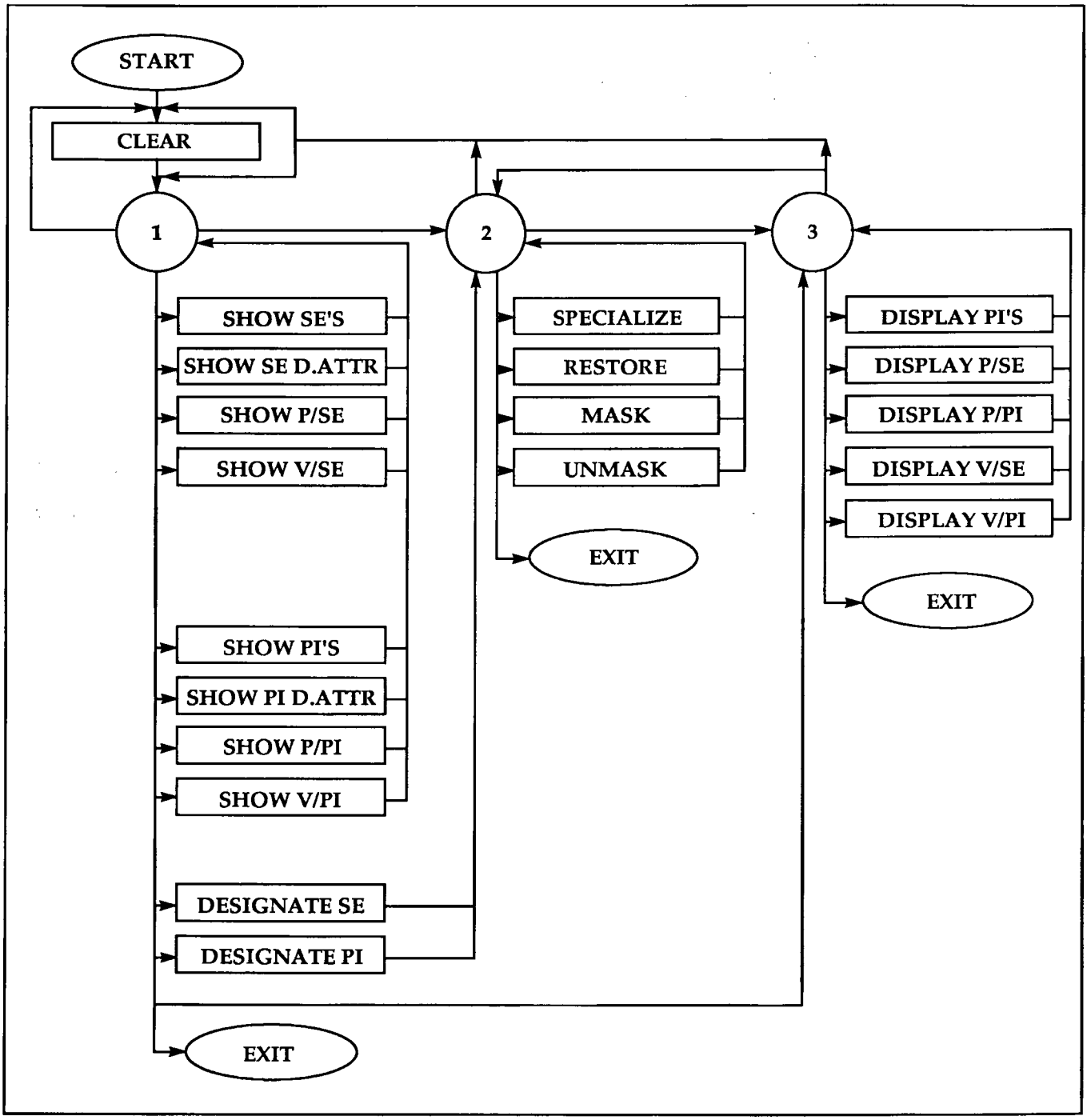

Figure 2. Structure of the Query Subsystem

The SHOW/DESIGNATE menu provides the means for exploring which SE's and PI's exist within the NEMS database and for examining the specifications for any particular SE or PI. The latter activity implies a means for "designating" (identifying) a specific SE or PI that is of special interest. This capability is provided as one of the SHOW/DESIGNATE options. A brief explanation of the various options under the SHOW/DESIGNATE menu is given below:

(a) list the names of all the SE's known to the system;

(b) view the defining attributes of some specific SE;

(c) view the parameter specifications of some specific SE;

(d) view the variable specifications of some specific SE;

(e) list the names of all the PI's known to the system;

(f) view the defining attributes of some specific PI;

(g) view the parameter specifications of some specific PI;

(h) view the variable specifications of some specific PI;

(i) designate a particular SE to be used in subsequent queries

(j) designate a particular PI to be used in subsequent queries

The concurrent designation of an SE and a PI is subject to an implicit compatibility constraint. This constraint has its basis in the fact that every PI is bound to some specific SE (its "master") and only compatible pairs in this sense may be designated. Any attempt by the user to violate this constraint is intercepted.

The designation of an SE or a PI or both, is interpreted as an indicator of the user's interest in some subset of the experiments logged within the database; namely, those that are associated with the designated modules. This subset is called the "active-set" and, at any stage of the query activity, it contains some subset of the tuples stored in the relation EXP-Lib. The number of tuples in this subset is continually displayed while the query subsystem is active. Upon first entry into the query mode the displayed value is the total number of experiments stored in EXP-Lib because no SE or PI have yet been designated. After designating a specific SE, the active set is contracted to include only those experiments where this specific SE is used. When a PI is also designated, the active set is further contracted to include only those experiments where the designated SE is used in conjunction with the designated PI. Note also that at any stage of the query activity, the user can choose the CLEAR option which "resets" any designated SE or PI. In this circumstance the active set is reset to include all experiments within EXP-Lib.

Once an SE or a PI has been designated, the user has the option of moving to the REFINE menu (recall that the designation of a PI implicitly designates an SE; hence there is always a designated SE at this point). The REFINE menu provides options both to enable the user to focus his interest on some specific subset of experiments and to control the extent of the output produced when these experiments are referenced. These two categories of activity are referred to as specialization and masking, respectively.

The specialization activity has relevance to both parameters and variables. An appropriate dialog allows the specification of either a specific value or a range of values for any particular parameter or variable within a designated SE or PI. The effect of such a specification is to contract the active set to contain only references to those experiments which are consistent with the specifications.

Frequently, during the query activity, there is no interest in actually viewing the values of all the parameters or variables that are recorded for the experiments. Masking provides a means to "hide" an arbitrary number of these parameters and variables so that during subsequent displays, their values are not presented. The user is thus able to focus his attention on those parameters and variables that are of special interest. Note that the masking operation does not affect the active set.

Both the specialization and masking operations may be "undone". In the REFINE menu, the user is provided with options either to retract previously specified value/range specifications (the restore option), or to unmask all the masked parameters or variables (the unmask option). When the user chooses the restore option, NEMS restores to the active set the references to those experiments that NEMS removed during the specialization process. The unmask option permits the user to unmask identifiers by category type; *i.e.*, all SE parameters, all SE variables, all PI parameters or all PI variables. (The current version of NEMS does not provide the facility to unmask individual identifiers.)

The DISPLAY menu is accessible whenever an SE or a PI has been designated (as noted previously, this implies that an SE will always be designated at this point). The intent here is to allow the user to browse through a portion of the experimental results that are stored in the NEMS database. In particular, the data that can be referenced corresponds to the active set together with the influence of the masking operations that are in effect.

Within the DISPLAY menu, the user has the following options:

(a) list all the PI's that have been used with the currently designated SE;

(b) display the active values assigned to the unmasked SE parameters, for each of the experiments in the active set (these will correspond to the experiments carried out with the currently designated SE and the currently designated PI (if any), appropriately modified by the user's refinement specifications);

(c) display the active values assigned to the unmasked PI parameters, for each of the experiments in the active set;

(d) display the values acquired by the unmasked SE variables, for each of the experiments in the active set;

(e) display the values acquired by the unmasked PI variables, for each of the experiments in the active set.

Even with the restricting facility provided by the "refine" option, there may still be a large number of experiments referenced in the active set. Some additional flexibility in browsing through these is provided by allowing the user to skip over an arbitrary number of these experiments. After displaying the appropriate data for each experiment, NEMS displays the total number of experiments which is yet to be displayed, and prompts the user for the number of experiments that he may wish to skip. Since the experiments are displayed in chronological order, this feature provides a means for accessing the most recently-carried out experiment within the active set.

The Delete Mode

The delete mode provides the user with the means to remove from the underlying database all references to any particular solution engine or problem instance. From the user's perspective, the procedure is straightforward since all that is required is the name of the SE or PI that is to be eliminated. The deletion operation can, however, result in extensive activity within NEMS, particularly in the case of the deletion of an SE. In this circumstance, NEMS deletes not only every experiment that is associated with the SE, but also every PI that has this SE as its master. The deletion activity includes both the removal of tuples from various relations as well as the removal of complete relations in the case where the SE/PI being deleted has vector valued parameters and/or variables associated with it.

Illustrative Application

The solution engine and problem instance concepts formulated in this study have been relatively loosely structured. The intent is to establish an intuitively simple conceptual framework which lends itself to a broad range of specific interpretations. The discussion in this section provides an example of one such interpretation within a simulation context.

Our example is based on a hypothetical continuous system simulation language called (say) SIMCON. In particular, suppose that SIMCON is to be used by the engineering department at the Apex Corporation to evaluate a control system design for a chemical process. Three separate process models based on different underlying assumptions, have been developed and there is an interest in evaluating the control concept using each of the three models.

Like all continuous system simulation "languages" SIMCON is in fact a software system that includes both a language processor and a set of service routines. Its general role is to "process" a problem specification (simulation model) written in the language syntax supported by SIMCON (this frequently involves a preliminary translation step which transforms the specification into a secondary language *e.g.*, Fortran, which becomes the execution vehicle). The problem specification in this environment would typically be referred to by the user as a "SIMCON program". The user "solves" the problem by directing an existing SIMCON executive routine to process a file that contains the SIMCON program (problem specification) that is under investigation.

Within the NEMS context, the SIMCON processor is regarded as a solution engine. This SE solves problems of a particular class; namely continuous system simulation problems that have been characterized in terms of program statements that are syntactically acceptable to the SIMCON language processor. Each such characterization (which exists as a unit of SIMCON code) corresponds to a problem instance. In our example scenario, there exist three such problem instances; these correspond to the three different "process plus control" models that are under consideration.

A prerequisite for the execution of the SIMCON processor is the specification of values for various parameters that are embedded in this SE. These parameters fall into two categories. In the first, are the parameters that directly influence the solution mechanism used by SIMCON to carry out its processing function. Representative examples included here are:

METHOD: This parameter serves to specify which of the available integration methods will be used to solve the ordinary differential equations within the model.

ABSERR: The step size adjustment mechanism associated with the variable step size integration methods within SIMCON seeks to maintain an estimated local truncation error close to a value that is controlled by two user-specified parameters. ABSERR is one of two parameters which control this value.

RELERR: This is the second parameter used in establishing the target value for the local truncation error.

HSTRT: This parameter provides the initial stepsize for the integration procedure. It is subsequently automatically adjusted as necessary.

The parameters in the second category also influence the processing function of SIMCON and must have assigned values. These parameters, however, are distinctive because their values originate as intrinsic features of the specific problem being solved; *e.g.*

N: This parameter provides the number of first order differential equations within the continuous model that is being studied. The three process models in the scenario under consideration have a different value for this parameter.

FINTIM: The right-hand end of the solution interval over which the problem is to be solved. (The left-hand end has a pre-assigned value of zero).

XZ: This is a vector valued parameter which provides the initial conditions for the differential equations of the model being studied.

Within the NEMS context, this latter category of parameters corresponds to reflected parameters whereas the first category corresponds to non-reflected parameters

A fundamental output provided by SIMCON is the sequence of values that results from the solution of the model equations; *e.g.*, the values on the solution trajectories of the state variables. Within the NEMS context, this output is a cluster list (for definiteness, suppose the identifier TRAJECT has been chosen to denote this c-list). We note, furthermore, that it should more properly be regarded as a feature of the problem being solved rather than as a feature of SIMCON itself. In other words, TRAJECT is a reflected c-list variable of SIMCON.

Another likely output provided by SIMCON is the average integration stepsize used over the solution interval, for any particular execution of SIMCON. This value (suppose it is maintained in a variable called AVG) will generally vary with the integration method being used and with the solution accuracy resulting from the values assigned to the parameters ABSERR and RELERR. In our assumed scenario, the value of AVG will provide some insight into the relative performance of the various integration methods that will be used in studying the behavior of

three different system models. This variable, AVG, is interesting because it can be regarded either as a reflected or non-reflected variable of SIMCON (*i.e.*, does its value represent a feature of SIMCON or of the problem being solved?). Such options can frequently arise and their resolution depends simply on the perception of the NEMS user.

In the NEMS context, our assumed interest in three different process models, corresponds to the creation of three different problem instances (the "owner" of each is, of course, SIMCON). The parameters of these problem instances will include the reflected parameters that are inherited from SIMCON. In other words, when the PI's are being CREAT'ed, NEMS will ensure that each automatically inherits the parameters N, FINTIM and XZ. Some of the PI's may of course, have additional parameters and these can be specified during the CREATE activity.

An (output) variable associated with each PI is the reflected variable TRAJECT that is inherited from the SIMCON solution engine. One can furthermore assume that the underlying design activity, in our assumed scenario, is concerned with system performance and correspondingly, a performance criterion would naturally be associated with the process models being studied. If the value of this criterion function is stored in a variable called PERFORM, then PERFORM would correspond to a second variable associated with each of the PI's and would, accordingly, be specified during the CREATE activity. It is interesting to observe here that one could use the reflected variable feature of NEMS to circumvent the need to explicitly specify the variable PERFORM for each of the PI's. With this alternate approach, PERFORM would simply be specified as a reflected variable of the solution engine SIMCON.

The above discussion provides some indication of how an intended experimentation activity with NEMS would be conceptualized. As suggested in this discussion, the first step is to characterize the solution engine and problem instances that are to be used, together with their respective parameters and variables. The NEMS create mode would then be used to introduce this information into the storage structures of the NEMS database.

The NEMS execute mode provides the service for carrying out the experimentation activity itself. To a large extent, the activity in this mode is driven by the characterizations acquired during the create phase. The first step is to obtain from the user, the values for the parameters within the modules that are to participate in the experiment (*e.g.*, values for **METHOD, ABSERR, RELRRR, HSTRT, N, FINTIM, XZ** and possibly others). The execute mode also provides the framework for actually carrying out the experiment execution and thereby generating values for the variables within the participating modules (*e.g.*, **TRAJECT, AVG, PERFORM,** etc). These values are automatically stored in appropriate NEMS storage structures (as outlined in the section on The NEMS Storage Structures).

After multiple experiments have been completed, the query mode gives the user the means for browsing through the accumulated experimental data to acquire whatever relevant information may be sought (e.g., is the control system performance influenced dramatically by the process model being used?).

The illustrative overview of this section has intentionally avoided the consideration of many important implementational aspects. These are primarily programming details which ensure that:

(a) The SE/PI code that is of interest in any particular experiment can be activated under NEMS control (i.e., the creation of an appropriate command file), and

(b) The SE/PI code incorporates the necessary read/write operations that properly access the NEMS files that are used for the interchange of data.

These details are outside the intended scope of the presentation.

Concluding Remarks

The NEMS software system described in this paper provides a productivity tool for researchers, software developers and system designers who have the need to carry out extensive sets of numerical experiments. The simulation community represents one such potential user group. The tool has been designed to facilitate both the execution of the experiments and the management of the results that are generated from these experiments. A query mode is provided which enables the user to browse through the accumulated results in order to acquire insights and determine directions for further experimentation.

The design of NEMS is based on a model of the problem solving activity which consists of three basic entities; namely, solution engines, problem instances and experiments. Each of these is characterized by a set of attributes which provide the basis for the design of a set of relations in a relational database. The vehicle used in the present NEMS implementation to set up and manage this database is dBase III. The attributes and the associated relations have been carefully formulated so that a reasonably broad class of numerical experimentation activity can be accommodated.

From the user's perspective, NEMS operates in four functional modes. These are (a) the create mode which allows the user to create solution engines and problem instances (i.e., insert into the NEMS database instances of both SE's and PI's); (b) the execute mode which provides the means for executing a compatible solution engine/problem instance combination thereby creating an instance of an experiment object; (c) the query mode which provides a convenient means for browsing through experimental results; and (d) the delete mode which allows the user to delete solution engines or problem instances from the underlying database. The create activity is based on the use of a set of template relations which provide the building blocks from which the database representation of each new solution engine or problem instance is formulated. A standardized dialog during the execute activity not only facilitates parameter value assignment but also ensures that assigned values are consistent with pre-specified range constraints. The query mode provides a mechanism that allows the user to specify a "viewing filter" which greatly assists in rapidly focussing on pertinent results. The delete mode is essentially a housekeeping facility.

An important and technically feasible enhancement to NEMS would be the extension of its role from a passive support tool to active support tool by incorporating goal

directed features. For example, in any particular installation, it is likely that NEMS will provide access to a large variety of numerical software packages (*i.e.* solution engines). A valuable addition therefore would be an advisor subsystem that would provide guidance to the user in choosing the solution engine that most closely matches his specific problem, or where applicable, explaining the relative advantages of several equally applicable engines. An interesting discussion of possible software systems structured along these lines has recently appeared (Abelson, *et al.*, 1989).

References

Abelson, H., Eisenberg, M., Halfant,M., Katzenelson, J., Sacks, E., Sussman, G.J., Wisdom, J. and Yip, K., May 1989. "Intelligence in Scientific Computing", *Comm. ACM*, pp. 546-562.

Korn, G.H., December 1985. "A New Interactive Environment for Computer-Aided Experiments", *SIMULATION*, vol. 45, no. 6, pp. 303-305.

Ören, T., and Zeigler,B., March 1979. "Concepts for Advanced Simulation Methodologies", *SIMULATION*, vol. 32, no. 3, pp. 69-82.

Standridge, C.R., September 1981. "Using the Simulation Data Language (SDL)", *SIMULATION*, vol. 37, no. 3, pp. 73-81.

Standridge, C.R., October 1981. "The Simulation Data Language (SDL): Applications and Examples", *SIMULATION*, 1, vol. 37, no. 4, pp. 119-130.

Standridge, C.R., and Sale, M.L., 1985. "A Tutorial on TESS: The Extended Simulation System", Proc. *1985 Winter Simulation Conf.*, pp. 73-79.

Standridge, C.R., and Pritsker,A.A.B., 1987. *TESS: The Extended Simulation Support System*, Halsted Press, New York.