Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

DELAB - A Simulation Laboratory

Miron Livny

Department of Computer Sciences
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706

ABSTRACT

DELAB is a simulation laboratory designed to provide support to programmers who build complex simulation programs and to system analysts who use these programs. In this paper we present the structure of the laboratory and report on the current status of the effort to implement it. The laboratory has been implemented in a 'bottom up' fashion. First we have developed the DENET simulation language which is a Modula-2 based discrete event simulation language. Once the language became operational, a database management system was added to the laboratory. For each simulation study a relational database is automatically created. When a simulation terminates it stores a description of the run in the database. The system analyst can later retrieve this data by means of a relational query language. DENET has been successfully used in a number of real life simulation studies. The database management system is currently evaluated by a number of researchers in our department who employ it in their simulation studies. The requests, criticism, and encouragement provided by users of both the language and the management system have guided our iterative effort to design and implement an effective simulation laboratory.

1. INTRODUCTION

Modern processing and manufacturing systems are characterized by a strong interdependency between their components. Consequently, performance analysis of such systems almost always requires a simulation study. The size and complexity of these systems make such a study a compound process. A need thus arises for a *simulation laboratory* that supports the construction of complex simulators and the management of long term simulation studies. DELAB is a simulation laboratory currently under construction, that aims to provide programmers and system analyist with such support. In addition to a powerful and modular simulation language, the laboratory will include a data management support system, an execution manager, and a set of output analysis utilities. In this paper we present the design philosophy of the laboratory and report on the current state of the DELAB project.

Guided by a desire for a better understanding of the needs of simulation studies and the observation that none of the existing simulation languages meets our requirements, we decided to take a 'bottom up' approach to the design and implementation of DELAB. One of the advantages of this approach is that it enables us to experiment with partial implementations. Prototypes of the laboratory that support only a limited set of functions can be tested in real life studies. The experience gained from the usage of lower level elements can be used to guide the design of higher level elements and to improve the implementation of the tested elements. We believe that such experience is crucial to the design of an effective simulation laboratory.

Since the lowest layer of a simulation laboratory is the simulation language, the goal of the first phase of the project was to design and implement a simulation language. The first phase of the project ended in the summer of 1985 when the DENET (Discrete Event NETwork) simulation language became operational. The language is based on the concept of Discrete Event System Specifications (DEVS) (Zeigler 1976). It views the simulator as a Directed Graph where a node represents a DEVS and a directed arc represents a coupling between two DEVSs. In the past two years the language has been used in a number of real life simulation studies. It was used to simulate distributed processing environments (Chang and Livny 1986), communication protocols(Qu, Landweber and Livny 1985), and production lines. A number of tools have been developed around the language. All tools adhere to the same modeling methodology and thus constitute a cohesive simulation environment. A language for distributed workload specification has been implemented and interactive debugging tools have been developed. Utilities for 'post mortem' analysis of traces generated by the reporting facility of DENET have been designed.

We are currently in the second phase of the DELAB project. In this phase we have been addressing the data management problem. A Database Management System (DBMS) that meets the special needs of a simulation laboratory was implemented. The DBMS can store descriptions of simulation runs that were generated by DENET and provides easy access to the stored data. The special data modeling needs of a simulation laboratory are currently studied in order to identify a data model that will meet the needs of such a

laboratory.

In the next section we present an overview of the structure of DE*LAB*. Section 3 presents the main features of DE*NET*. The current status of the DBMS is discussed in section 4. Finally, conclusions are stated in Section 5.

2. The Structure of DE*LAB*

Four main components can be identified in the current design of DE*LAB*: a *programming environment*, an *execution manager*, a *data manager*, and an *output analysis environment*. Each of these components provides support for a different type of activity. DE*LAB* was designed to support the activities of programmers who build simulators and system analysts who use them. Programmers need support in mapping a discrete event model to a program, in debugging it, and in verifying the implementation of the model. Once the program has been debugged and verified it is transferred to the system analyst who runs the simulator and evaluates the results. At the data gathering stage of a simulation study support is needed in selecting values for input parameters, in utilizing available computing resources, and in storing the data. The last, and in most cases the most exciting, stage of a simulation study is the results analysis stage. At this stage the system analyst needs tools for data retrieval, statistical analysis, and graphical display.

A block diagram of DE*LAB* is presented in Figure 1. The programmer interacts with the simulation laboratory via a programming environment, whereas the system analyst interacts with the laboratory via an execution manager and an output analysis environment. The programming environment that includes a powerful simulation language, an interactive debugger, and a flexible tracing and reporting facility, supports the programmer in coding the simulator. The simulator is automatically interfaced with the database management system (DBMS) and the execution manager. Via the execution manager the system analyst assigns values to simulation parameters and assigns runs to computers. We view the simulation laboratory as operating in a multi computer environment. In such an environment the allocation of runs to machines is not a trivial task especially when some of the machines are not available all the time; see Mutka and Livny (1987). One of the services provided by the execution manager will be *remote execution* of simulation runs. The manager will assign waiting runs to available resources, monitor their progress, and restart them in case of a failure.

One of the significant lessons we have learned from the simulation studies we have conducted is the vital importance of a database management support system. The amount of data generated throughout the lifetime of a simulation study is immense. Tens of attributes are required to define an experiment and hundreds of values are generated by a single run. Keeping the results in a file system is a nightmare since after a short period of time there is no way to map the encrypted file names and directories to experiments. At that point, the study becomes highly vulnerable to errors in relating results to experiments and a DBMS becomes essential.

Two users who share the same database do not necessarily share the same view of the data stored in the database. When more then one view has to be supported by the DBMS, a mapping from the different views to a single schema of the data has to be defined (Korth and Silberschatz 1986). The DBMS of DE*LAB* is designed to support two different views: the *system view* and the *simulation view*. In the course of the different simulation studies we realized that we use both views to describe an experiment. The first view is used when we argue about the different runs, whereas we use the latter view

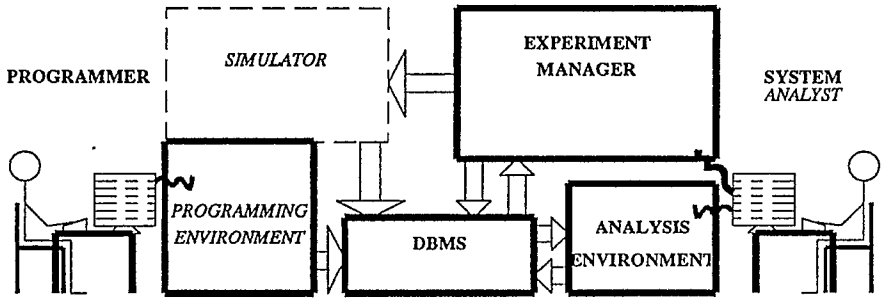

Figure 1: Structure of DE*LAB*

to define the experiment in terms of the primitives provided by the simulation language. Since the system analyst accesses the data via the execution manager and the analysis environment, the interfaces between these elements and the DBMS are based on the system view. Data generated by the simulator is described in terms of the simulation view.

The analysis environment will include utilities to support the statistical analysis and graphical display of data. The current design of DELAB does not include facilities for decision support or model management. Although we view these facilities as very important it was decided not to include them in the current design. We plan to address the issues of model management and decision support in the context of simulation studies once we have gained experience from using DELAB in real life simulation studies. We trust that such experience along with the insight that will be gained by current efforts to study these issues (Balci 1986 ; Balmer and Paul 1986) will enable us to design effective model management and decision support systems for DELAB.

3. The DENET Language

DENET is a discrete event simulation language built on top of the general purpose programming language Modula-2 (Wirth 1983). The language is designed to meet the needs of system designers and performance analysts who face the problem of predicting the behavior of complex parallel systems. The data structures, control structures, and timing mechanism of DENET constitute an efficient tool for simulating the behavior of compound processing and manufacturing systems. DENET provides a powerful and highly modular simulation environment. It is a successor of the DISS simulation language (Melman and Livny 1984).

The DENET language is based on the Discrete Event System Specifications (DEVS) modeling methodology. The design of the language was guided by the desire to develop a simulation language that will follow the modeling concepts of this methodology as closely as possible. Discrete Event Systems Specifications were first formally defined by Zeigler (Zeigler 1976). In (Livny 1984) we have extended the definition of a DEVS to include *ports* and introduced a new definition for Discrete Event Networks. The port structure enables us to represent a coupling between two DEVSs as a mapping from the output port of one DEVS to the input port of the other. Such a mapping preserves the autonomy and structural independency of the two systems and thus leads to modular and extensible models. A new model can be formed from an existing one by adding, removing or replacing a DEVS. Furthermore, it provides the means for maintaining structural similarity between the simulated system and the discrete event model. Topological changes in the simulated system are easily mapped to changes in the layout of the Discrete Event Network. Structural similarity improves the effectiveness of communication between the designer and the modeler consequently enhancing the model verification process.

DENET carries this similarity one step further. Following the modeling methodology that has emerged from the concept of DEVS, it provides the means for maintaining structural similarity between the simulated system and the simulator. The Discrete Event Systems methodology constitutes the formal foundations of DENET. Each of the elements of a Discrete Event Network specification can be directly related to an element of the language. Discrete Event Systems are mapped to Discrete Event Modules (DEVM). The coupling between DEVSs is represented by a Discrete Event Coupler (DEVC). The DEVC captures the uniqueness of discrete event simulation and thus differs from interfaces employed by general purpose languages. Unlike other parallel simulation languages that are message based (Bagrodia, Chandy and Misra 1987; Schwetmann 1986), the sole synchronization primitive of DENET is the *event*. Synchronization between two DEVMs is viewed as an event triggered by the source DEVM and observed at a given instance by the destination DEVM. Countdown Clock state variables (Zeigler 1976) are also realized by events. DENET distinguish between *internal* events and *input* events. When a Countdown Clock expires it triggers an *internal* event, whereas events triggered by one DEVM and observed by another are defined as *input* events of the latter. All events, whether triggered internally or externally, are treated in the same way and thus constitute a cohesive timing mechanism.

The properties of the DEVM and DEVC make the establishment of a library of DEVMs a natural and simple task. A number of DENET DEVMs, each of which represents the behavior of a different Discrete Event System, can be easily grouped to form a library. A DEVM stored in the library can represent, for instance, a local area communication network, a machine tool, or a computer system. When an implementor of a new model realizes that the specifications of one of the models' DEVSs is met by a library DEVM, the DEVM can be easily incorporated in the new model. At any time, a simulator can be constructed dynamicly from private DEVMs and library DEVMs. As new DEVSs are realized by DEVMs, they can be added to the library.

Since there is no structural dependency between two DEVMs any pair of DEVMs can be connected by a DEVC. Each DEVM is a self contained compilation unit and therefore there is no need to expose the internal structure of the DEVM to potential users. DENET provides the means to dynamicly construct a simulator from a set of executable presentations of DEVMs and DEVCs. Consequently, the executable image of

a First In First Out server, an ETHERNET local area network, or a Store and Forward communication processor can be shared by many users. They can use these generic modules whenever the functionality of the module meets their current needs.

DE*NET* views the simulator as a Directed Graph (D-Graph). Figure 2 depicts a simulator of a Distributed Database Management System (DDBMS). Each node of the graph is an instance of a Discrete Event Module (DEVM) which is a self contained module that realizes a Discrete Event System Specification (DEVS). DE*NET* furnishs a wide range of data structures and event management primitives that support the realization of a Discrete Event System by a DEVM. For instance, in the DDBMS example the Optimistic Distributed Concurrency Control algorithm is realized by one DEVM (*OptCCManager*), while the arrival process of transactions and the communication network are realized by a second and third DEVMs (*source*, and *Network* respactively). The complexity, and thus the size, of an DEVM can vary from one module to another. More than 1000 lines of DE*NET* code were required to realize the logic of the Optimistic algorithm. However, less than 300 lines were needed to describe the arrival process of transactions, and only 112 lines of code were used to realize the model of the communication system.

A node has an input port associated with each incoming arc and an output port associated with each outgoing arc. An arc in the simulator represents a mapping from the output variables that constitute the output port of the source node, to the input variables and events that constitute the input port of the target node. Nodes synchronize their activities via arcs. An arc can be viewed as a multi-wire cable that connects two sets of memory locations. One set is owned by the source node and the other one is owned by the destination node. The connection is uni-directional; each wire goes from a source location, which is an instance of an output identifier, to a destination location, which is an instance of an input identifier. When a value is assigned to an output variable, the assignment is propagated via the arc to the input identifier to which the given output variable is mapped via a wire. If the target node considers such an assignment an event, it is invoked as a result of the assignment. For instance, when one ResourceManager of the DDBMS wants to transfer a message via the communication network, it assigns the address of the *entity* that describes the message to one of its output variables. Since this output variable is mapped to an input event in the network node, the assignment triggers an event at that node. The network node 'awakens' and performs the activity associated with a request to transfer a message.

The arcs of the simulator are instances of Discrete Event Connectors (DEVC). A DEVC is a mapping from output identifiers to input identifiers. Each DEVM has a set of output identifiers and input identifiers through which it interacts with its surrounding DEVMs. Each input port is an instance of the node input identifiers and every output port is an instance of the node output identifiers. The simulator is constructed dynamically from a given set of DE*NET* DEVMs and DEVCs. DE*NET* can be viewed as consisting of two sub-languages - a low level and a high level language. The lower level language is used to describe DEVMs and DEVCs, whereas the higher level language is used to define the simulator. The structure of the simulator is given in a special file called the *topology* file. Figure 2 depicts the layout of a DDBMS simulator. This instance of the simulator consists of 65 nodes and 32 arcs. It models the behavior of a 16 site DDBMS interconnected by a broadcast communication channel. The topology file that describes the structure of the simulator is presented in Figure 3. An interpreter is invoked at execution time to read the topology file, to translate the description, and to build the simulator. Note how easy it is to change the number of sites in the simulator or to replace the Optimistic algorithm with another Distributed Concurrency Control algorithm. Two numbers have to be changed in the first case and only one identifier, the name of the DEVM that realizes the algorithm, has to be modified in the second case. Without much effort, a menu driven simulation environment can be established using DE*NET*. In the case of the DDBMS example, once a set of Distributed Concurrency algorithms was realized by DEVMs, a menu driven environment for the study of this type of distributed algorithms can be established. The modularity of DE*NET* and the flexibility of the language used to define the structure of the simulator turn DE*NET* into a 'production line' for menu driven simulation environments.

DE*NET* is based on a unique view of events. In a discrete event system state transitions take place only upon the occurrence of an *event*. When an event occurs the current state is evaluated and a new state is established. An event is represented in DE*NET* by an identifier that has a *type*, a *value*, a *state*, and an *activity* associated with it. Unlike the type attribute which is static, the value, state, and activity are dynamic properties of the event. Whenever a value is assigned to an *event variable* its state is checked. If the event is in the *WaitFor* state, i.e. the DEVM is waiting for an assignment to take place, the module is invoked and the activity executed. The *waituntil* statement, which is the main element of timing mechanism of DE*NET*, operates on a set of events. It associates an activity with each event and suspends the execution of the DEVM until one of the events is triggered. Once an event is triggered, the activity associated with it is executed and control is transferred to the statement that follows the waituntil statement. The activity associated with an event represents the state transition logic of the event at the given

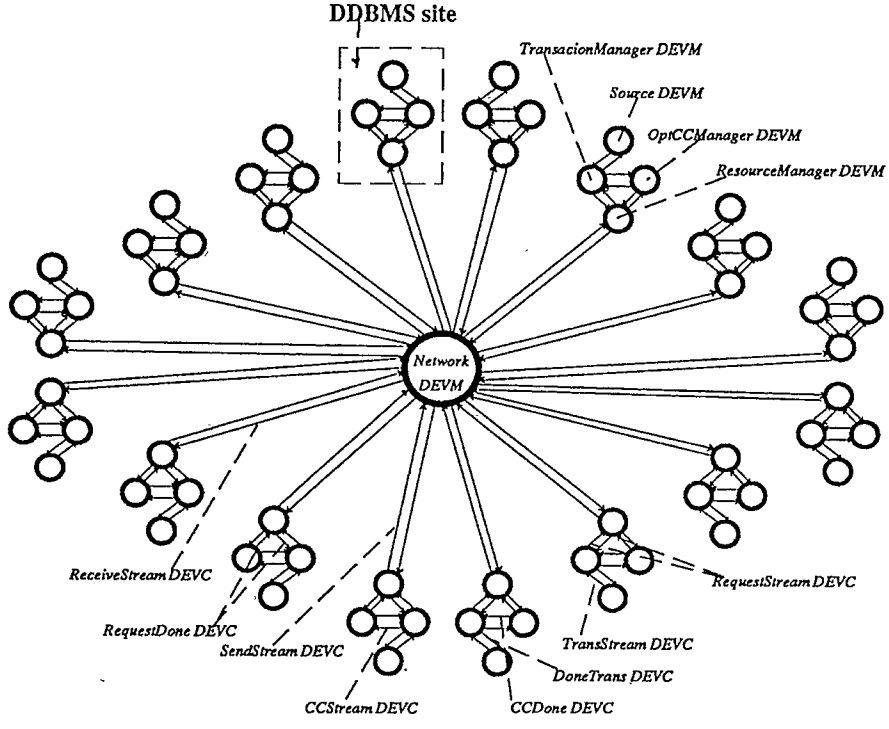

Figure 2: An example of a DE*NET* Simulator
(A Distributed Database Management System (DDBMS))

```
1 := network ; (* node 1 is the network node *)
(* define the sites *)
FOR i := 1 TO 64 BY 4 DO
        i+1 := Source ; (* first node is the source of transactions *)
        i+2 := TransactionManager ; (* second node manages transactions *)
        i+3 := OptCCManager ; (* third node is the CC manager *)
        i+4 := ResourceManager ; (* last node allocates resources *)
        [i+1 | i+2] := TransStream ; (* connects the source to the transaction manager *)
        [i+2 | i+1] := DoneTrans ; (* connection for completed transactions*)
        [i+2 | i+3] := CCStream ; (* connection for CC requests *)
        [i+3 | i+2] := CCDone ; (* connection for CC replies *)
        [i+2,i+3 | i+4 ] := RequestStream ; (* for resource requests *)
        [i+4 | i+2,i+3] := RequestDone ; (* for completion notices *)
END ;
(* Connect the sites to the network *)
FOR i := 5 TO 65 BY 4 DO
        [i | 1 ] := SendStream ; (* connect resource manager to network*)
        [1 | i ] := ReceiveStream ; (* connect network to resource manager *)
END ;
```

Figure 3: An example of a DE*NET* topology file

instance. A change in the state of the system may entail a change in the transition logic of the event. The waituntil statement of DEVNET supports such dynamic changes.

DEVNET includes utilities that support the creation and manipulation of *entities* and *queues*. The queue data structure is a widely used structure and has a number of basic operations associated with it. Another structure supported by DEVNET is the *probe* structure. Probes are used for data sampling. Time weighted sampling and batch sampling are supported along with means to derive statistics for the values sampled.

The DEVNET environment consists of a compiler, a set of Modula-2 definition modules, and an object file that contains the runtime support of the language. In a three step process, DEVNET modules provided by the user are turned into an executable file. Using the same executable file the user can construct different simulators, run them with a variety of input parameters and control the tracing information generated by each run. Figure 4 summerizes the structure of the environment and the flow of activities involved in compiling and running a DEVNET simulator. The three input files read by DEVNET at runtime enable the user to select the nodes and the arcs of the simulator, to assign values to the input variables of each node, and to control the debugging and reporting mechanisms of the language. The phases of the compilation process are transparent to the user unless the user wants to see them.

Each Discrete Event Module type is represented by an execution module and an optional definition module. DEVNET modules are compiled into Modula-2 by the DEVNET compiler. A definition module is created by the compiler for each translated module. The compiler is written in Pascal and has an error correction mechanism. Each DEVNET module is compiled separately into a Modula-2 module that is compiled, in turn, by the Modula-2 compiler into an object file. Therefore, when a DEVM is modified, only one file has to be recompiled. There is no need to update the object file of modules whose DEVNET code was not changed.

DEVNET is currently running on VAX computers and SUN workstations. On the VAX it runs under both BSD UNIX 4.3 and VMS. The language can be easily ported to any combination of processor and operating system that have a Pascal and Modula-2 compiler.

4. The Data Management System

When an analytical presentation of the relationship between the properties of a system and its behavior cannot be derived, the system has to be simulated in order to predict its behavior. The simulation program that realizes a model of the system captures this relationship and thus can be viewed as a function that maps a system specification to a quantitative behavioral profile. Each simulation run relates such a profile to a set of simulation parameters that defines the system and its operational environment. Consequently, the outcome of a simulation study can be represented by a relation (table) (see Korth and Silberschatz 1986) on two compound domains: the *parameter* (P) domain and the *result* (R) domain. We refer to this relation as the experiment relation (EXP) of the study.

A run of the simulation is represented by a tuple (row) in EXP. The first column of a tuple represents the values assigned to the simulation parameters. These parameters control the properties of the simulated system and the environment in which it operates. Simulation parameters can be used to control the size of the system, its speed, the load imposed on the system, or the temperature in which it operates. The second column of each tuple represents a profile of the system behavior. A profile may consist of a histogram of response times, a report on system utilization, or a count of completed jobs. In most cases each of the two columns of the EXP relation is a compound entity that consists of a large number of attributes. For instance, if we simulate a processing system with 32 sites, where each site is characterized by 5 parameters and generates 10 result values, the first column represents the value of at least 160 simulation parameters. A profile of the simulated system, represented by the second column, consists of 320 numbers. Due to size of the entities represented by a tuple and the number of tuples generated in the course of a typical simulation study, the management of the EXP relation is a time consuming and error prone process. A need thus arises for a Database Management System (DBMS) that will ease the burden of managing the data generated in the course of a simulation study. A DBMS can provide means for storing the data in a database and utilities to efficiently retrieve it.

We have implemented a relational DBMS that meets the special needs of DEVNET. The compiler and runtime environment of the language have been instrumented to automatically generate a description of the run. Since the structure of the P and R domains depends on the simulated system and the experimental frame of the study (Zeigler 1976), the structure of the data generated by a simulation program varies from one study to another. The structure of these domains, in turn, determines the input/output structure of the simulation program that realizes the model employed by the study. Therefore, one relational scheme cannot satisfy the needs of all experiments. Moreover, it is very unlikely that the same database can be used by two studies. For each study a relational schema that captures the structure of its P and R domains has to be devised and a database that realizes this schema has to be created.

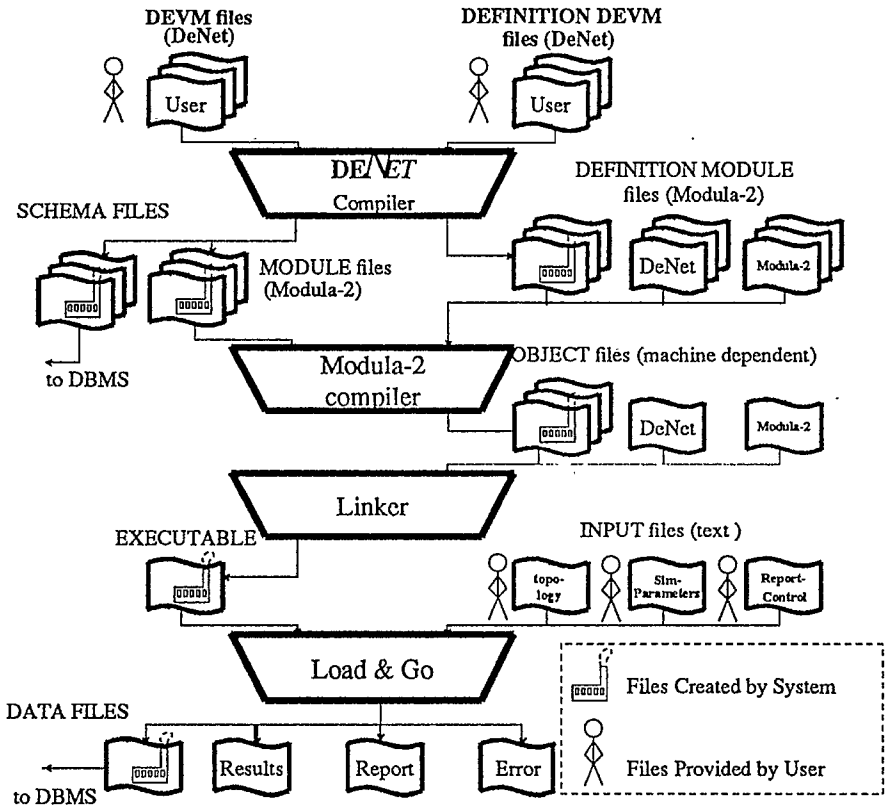

Figure 4: DE/NET Environment

DE/NET provides the means by which a simulation program can declare the structure of both its P and R domain. Since the language views a simulator as a Directed Graph it imposes a graph structure on the P domain. A run of a DE/NET program is characterized by the topology of the discrete event network and the values assigned to the *input parameters* of the nodes. Input parameters are special variables defined by the DEVM and used to personalize each of its instances. For each run the user provides a file that contains a description of the network and values for the input parameters. The runtime environment of DE/NET reads the description and the values from the file, builds the network of DEVM and DEVC instances, and assigns the values to the input parameters. Output parameters and probes are the means by which a DEVM declares its contribution to the R domain. Each element in R is a list of nodal results where each element in the list consists of the values of the output variables and probes of one DEVM instance. Each probe represents a sequence of samples preformed by the node. When a node terminates, the values of its output parameters as well as the mean, standard deviation, and count of its probes is automatically displayed.

All the information required to plan the relational schema (Korth and Silberschatz 1986) of a simulation is available to the DE/NET compiler. We have instrumented the compiler to automatically generate such a schema. The schema is passed to the DBMS that creates the relational database. However, the structure of the P domain leads to a database design which does not meet the criteria of first normal form. Since P is not an atomic domain not all the attributes of a tuple in EXP are atomic values.

Based on a schema generated by the compiler, a separate relational database is established for every simulation program. The relations that constitute the database can be classified in to three groups: a '*global*' group, a '*per DEVM*' group, and a '*per run*' group. The first group consists of two relations whos schema is common to all studies. The second group consists of subgroups each of which corresponds to a

DEVM type. Each subgroup consists of three relations that represent the input/output structure of the respective DEVM. The third group of relations represents the set of topologies that were used in the course of the study.

For each DEVM the compiler generates a file that contains a list of the names and types of all input parameters, output parameters, and probes declared by the DEVM. Following the first successful compilation of a DEVM, the DBMS reads the file and creates three relations that capture the input/output structure of the DEVM. Each time a DEVM is recompiled the DBMS checks whether the input/output structure was modified. When a modification is encountered, the structure of the relations associated with the DEVM is updated. Depending on whether an attribute was added or removed from a relation actions to guarantee the integrity of the stored data are performed.

When a simulation run terminates, the runtime environment of DENET passes to the DBMS a description of the topology of the discrete event network and the values of all input parameters, output parameters, and probes. The DBMS adds a new tuple the EXP relation and checks whether a run with the same topology was recorded. When a topology is used for the first time, a relation is created and tuples that capture the topology of the discrete event network are stored in it. The values of the input parameters, the output parameters, and the probes are stored in relations associated with each DEVM type.

A rich set of relational operations and aggregate functions is provided by the DBMS of DELAB. For example two projections and one join operation are needed to retrieve a table that relates the aggregated arrival rate of a system to its throughput. The database can be accessed via an interactive interface or by invoking a set of C procedures. The DBMS is written in C and it assumes a UNIX environment.

5. CONCLUSIONS

Conducting a simulation study of a modern processing or manufacturing system is a compound and time consuming process. It is not uncommon to find simulation studies that use large simulation programs, generate immense amounts of data, consume a large amount of processing capacity, and span many months of activity. The complexity of the simulated systems leads to the construction of large simulation programs that have to be debugged, verified and maintained over a long period of time. Each run of the simulation generates data that has to be stored on a mass storage device in a way that will provide efficient retrieval at a later stage. When a simulation study is executed in a multi-computer processing environment, the availability of resources has to be monitored in order to guarantee efficient usage of these resources.

The different activities involved in a simulation study raise a number of challenging problems. DELAB encapsulates our solutions to part of these problems. The methods and tools that constitute the simulation laboratory are based on techniques that were developed by a number of different disciplines. DELAB is an example for how advanced programming and management techniques can be translated into a cohesive set of tools that improve the efficiency and effectiveness of simulation studies. Although different disciplines have contributed to DELAB all the components of our simulation laboratory are based on a common modeling methodology. A prototype of the laboratory is operational and has been used in a number of real life simulation studies. The requests, criticism, and encouragement provided by colleagues and industrial users have guided the iterative design and implementation effort. However, much still remains to be accomplished before a simulation laboratory that covers all the aspects of a simulation study is available.

REFERENCES

Bagrodia R. L., Chandy M. and Misra J. (1987). A Message-Based Approach to Discrete-Event Simulation. *IEEE Trans. on Software Engineering* Vol. 13 No. 6.

Balci O. (1986). Requirements for Model Development Environments. *Comput. & Ops. Res.* Vol. 13, No. 1.

Balmer D. W. and Paul R. J. (1986). CASM - The Right Environment for Simulation. *J. Opl. Rec. Soc.* Vol 37, No. 5.

Chang H. and Livny M. (1986). Distributed Scheduling Under Deadline Constraints: A Comparison of Sender-initiated and Receiver-initiated Approaches. *Proceedings of the 1986 IEEE Real-Time Systems Symposium.*

Korth H.F. and Silberschatz A. (1986). *Database System Concepts.* McGraw-Hill, New York.

Livny M. (1984). The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems. Ph.D Thesis, Weizmann Institute of Science, Rehovot, Israel. (also available as Technical Report #570, Department of Computer Sciences, University of Wisconsin-Madison).

Melman M. and Livny M.(1984). The DISS Methodology of Distributed Systems Simulation. *Simulation* April 1984.

Mutka M. W. and Livny M. (1987). Scheduling Remote Processing Capacity in A Workstation-Processor Bank Network. *Proceedings of the 7th International Conference on Distributed Computing Systems.*

Qu Y., Landweber L.H. and Livny M. (1985). PARING: A Token Ring Local Area Network with Concurrency. *Proceedings of the 10th Conference on Local Computing Networks.*

Schwetmann, H.D (1986). CSIM: A C-Based Process-Priented Simulation Language. *Proceedings of the Winter Simulation Conference,* December 1986.

Wirth N. (1983). *Programming in Modula-2.* Springer-Verlag, New-York.

Zeigler B.P. (1976). *Theory of Modelling and Simulation.* Jhon Wiley & Sons, New York.

AUTHORS' BIOGRAPHY

MIRON LIVNY is an assistant professor in the Computer Sciences Department at the University of Wisconsin - Madison. He received a B.A in mathematics and physics from the Hebrew University in 1975, and an M.Sc. and Ph.D in computer sciences from the Weizmann Institute of Science in 1978 and 1984 respectively. He has been involved in a number of simulation based performance studies of existing and planned processing and communication systems. His current research interests include distributed resource management, communication protocols, and modeling and simulation techniques. He is a member of ACM and IEEE.

Miron Livny
Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton St.
Madison, WI, 53706,U.S.A
(608) 262-0856