Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

A FRAMEWORK FOR REUSABILITY USING GRAPH-BASED MODELS

Dana L. Wyatt

Department of Computer Science
University of North Texas
Denton, Texas 76203

ABSTRACT

This paper presents a framework for developing and managing reusable simulation components using graph-based models. Graph-based models can be defined as those models in which communications between submodel component are represented with a directed graph. A system architecture which readily supports this approach is presented and an example application domain is then outlined. Finally, potential research implications of this framework are discussed.

1. INTRODUCTION

Simulation is a popular technique used to assist decision makers in a wide variety of disciplines, including agriculture, business, engineering, and the social sciences. In 1989, it was considered of such dramatic importance to the technological needs of the United States that it was named one of twenty-two critical technologies by the Department of Defense and the Department of Energy [Pritsker 1989].

However, modeling and simulation is a time consuming and complex task. Used incorrectly, its results can be grossly misleading. In recent years, a considerable amount of time and energy has been placed in the examination and evaluation of various modeling methodologies [Balci 1989; Derrick et. al. 1989; Thesen 1989]. In fact, Derrick reports that no fewer than thirteen modeling methodologies are currently in use. Each methodology offers its own advantages and disadvantages. One is selected by a modeler in hopes that it can offer the appropriate perspective for a specific problem. The goal, of course, is to model a system correctly and consistently, using an appropriate level of detail.

Unfortunately, accurately modeling large and complex systems often proves to be quite difficult and expensive. Experienced modelers are in short supply [Balci and Nance 1987]. Thus, many simulationists have limited experience in modeling complex systems. Tools and techniques are available to assist the modeler in developing acceptable simulation studies. These include formal life cycle methodologies and simulation environments. However, little has been done to address the issue of reuse as it applies to simulation.

2. A FRAMEWORK FOR REUSABILITY

This paper will address the issue of simulation component reuse by examining a framework which supports reusability for graph-based models. This framework is described in general terms and has been utilized by implementations in several languages. It is presented in order to spark the reader to consider the implications of such an framework.

2.1 Reuse

Software reuse is a technique, as well as a philosophy, which offers some real benefits to the simulation community [Reese and Wyatt 1987]. It has been used with some success in specific commercial sectors of the software engineering community. Reuse exists at two levels: formal and informal. In the software engineering community, formal reuse includes the reuse of requirements specifications, design documents, and other high level life cycle documents. Informal reuse includes the reuse of data structures and code.

Advocates of reuse believe that building software from components which have been previously validated makes the task of constructing large software systems somewhat easier. However, effective reuse of components requires that an environment exist which makes them easily available and accessible. In addition, it requires an organizational philosophy which encourages and supports this "building-block" approach to software generation rather than the maverick "go-it-alone" approach. The commitment to reuse from an organization is not without cost. In many cases, the initial cost of developing generalized, reusable components exceeds the cost of developing the software in a traditional manner. However, the return on investment is hopefully achieved in subsequent projects when the reusable components may be included.

2.2 Simulation Environments

There has been some movement in recent years to design and implement some form of integrated software support environments for simulation [Balmer 1987; Reese and Sheppard 1983]. It is perceived that these environments will improve the productivity of modelers and increase the likelihood of successful simulations. However, the availability of support systems still lags far behind that found in the software engineering community.

In the software engineering community, the capabilities of support environments vary. The most comprehensive environments include support for virtually all areas of the software life cycle, from requirements specifications to detailed design and test plans, from code support to version control. Other environments might include only support for informal reuse aspects or part of the life cycle.

However, the simulation community is still somewhat fragmented over what functional components an integrated software support environment for simulation should include. Comprehensive environments might include support for:

- A particular simulation methodology
- Basic simulation activities (event and manipulation routines, random number generation, etc)
- Graphical model support (the ability to accept graphical specification of models and/or to produce graphical output)
- Automatic programming (automatic generation of simulation code from formal definitions of model behavior)
- The integration of artificial intelligence

Smaller environments might only include, for example, support for a particular methodology or for automatic programming.

The framework proposed herein could be considered a specification for an integrated software support environment because it provides the modeler with a system which supports the second through fourth items above, as well as providing an environment which supports easy access to previously defined model components, thus encouraging reusability.

2.3 Graph-Based Models

The models for which this framework is intended are those which can be described using directed graphs. This modeling approach has been used historically with some success [Kaplan 1987]. A directed graph consists of a collection of nodes, which correspond to model components, and a collection of arcs, which correspond to communication paths between nodes. The basic premise behind this framework is that model components can be developed, validated, and saved and that communication paths between selected components can then be specified in order to describe a complete model.

Figure 1 illustrates a graph-based model constructed from three nodes, N1, N2, and N3. Arcs A1 and A2 are stimuli arcs and have no input node. They are used to indicate that input from the outside world is passed on to nodes N1 and N2, respectively. Arcs A3 and A4 are connective arcs and indicate a communication path exists between the indicated nodes. Arc A5 is an output arc. It has no output node and corresponds to output from a node which is passed on to the outside world.

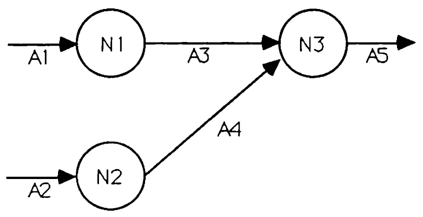

Figure 1. A Sample Graph-Based Model

Figure 1 describes a relatively simple model. However, by adding a scenario, the graph begins to describe more interesting models. It might represent a manufacturing system in which N1 is a machine which constructs widgets using raw material supplied through A1. N2 constructs widget caps using raw materials supplied through A2. N1 and N2 pass the items constructed to N3 along arcs A3 and A4, respectively, where they are then assembled at node N3 and then shipped through A5. Similarly, one could construct scenarios in which nodes represent processors in a multiprocessor environment or an unloading/loading process for ships in a harbor.

2.4 System Architecture

Graph-based models such as the ones described above may be developed and implemented using the graph-based model (GBM) system framework discussed in this section. Figure 2 illustrates the architecture of the GBM system, which is comprised of three subsystems: a component base and associated evaluator, a model base, and a simulation manager.

2.4.1 Component Base and Evaluator

The Component Base (CB) consists of a library containing all

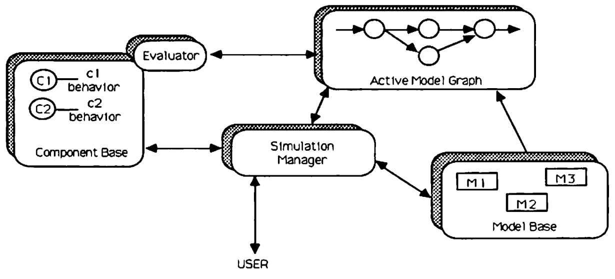

Figure 2. The GBM System Architecture

odel components which are available to the modeler. A model component corresponds to a node and is the basic building block of this system. It consists of a component name and a set of behavioral specifications describing ithe component's operation and output given a set input values. Components can be added to the CB as needed, and it is this structure which gives the system both power and flexibility.

The CB requires an Evaluator to determine the appropriate behavior for a component given a specific instance of input values. The Evaluator is an interpreter which retrieves the appropriate component from the CB library given its name, executes a behavioral specification script, and determines the appropriate output values and the delay through the component.

2.4.2 Model Base

The Model Base (MB) is a library in which complete models are stored. Models are created by specifying a directed graph consisting of components from the CB and communication paths between the components. Thus, a model is a collection of nodes and arc connections. The development of models in this environment may be accomplished graphically or textually, depending upon the user interface available. Once developed, a model must be loaded from the MB into the Active Model Graph area in order to simulate its behavior. It is the Active Model Graph area contains the model and the system environment for a particular simulation run.

2.4.3 Simulation Manager

The last subsystem is the Simulation Manager. The Simulation Manager is responsible for coordinating the simulation, maintaining the clock, and updating the event list. The Simulation Manager also functions as the interface between the modeler and the GBM system by providing utilities for creating, modifying, and deleting MB entries and CB components, as well as for controlling a particular simulation experiment.

2.5 Discussion

The GBM system as presented above consist of three subsystems which function collectively to provide a simulation environment for graph-based models. The Component Base and Model Base are essentially databases of model components. The only significant code is found in the Simulation Manager and Evaluator. In addition, the primary data structures in this system are the event list and the Active Model Graph.

The structure of the GBM system is such that virtually any system which may be represented by a graph-based model may be simulated. The Simulation Manager is a generalized control routine and needs minimal tailoring to be suitable for other problem domains. The Evaluator is tailored for a specification behavorial specification technique and must be rewritten if the technique changes.

Sources for incoming stimuli such as jobs in a manufacturing system or messages in a communication system may either be specified at run time as input or defined using a component whose sole purpose is to generate stimuli. Output is handled in a similar manner. Output arc values may be displayed at each time interval, or a component whose task is to simply record values may be placed as a terminator. This last technique would allow post-processing of simulation output for graphical display purposes or for statistical uses.

It is believed that this framework offers a significant potential for component reuse. By defining model components using a simple, building-block nature, a modeler may construct a larger, more complex model by specifying the relationships which exist between the components. In the following section, a research project at the University of North Texas which uses the framework will be discussed.

3. DIGITAL CIRCUIT SIMULATION

During the past year, a digital circuit simulator previously written in Lisp was reimplemented in C++ and C using the GBM framework described above [Benavides and Wyatt 1988; Ali and Wyatt 1990]. Digital circuit simulation is a natural application for any graph-based modeling system. It is a rather intuitive process to examine figure 3 and recognize that it is equivalent to the generic graph-based model described in figure 1. In figure 3, nodes N1 and N2 are defined to be 2-input AND gates and node N3 is defined as a 2-input NAND gate. The arcs (called nets) correspond to wires between the gates.

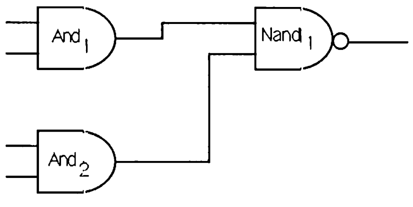

Figure 3. A Sample Digital Circuit

3.1 Active Model Graph Data Structure

The Active Model Graph area contains the current model which is being simulated. This model is represented as a directed graph of gates and nets. Because of the way in which gate and net information is required, each must be accessible directly from the simulation manager. Therefore, each gate entry in the model must include:

- gate name
- gate type
- list of input nets to the gate
- list of output nets from the gate

In addition, each net entry in the model must include:

- net name
- net value
- list of gates which drive it
- list of gates which it drives

This organization for data structures may be met in one of many manners. In the C++ implementation, classes are used. In the C implementation, threaded linked lists are used. The actual implementation structure is not important as long as the information is accessible.

3.2 Component Base and Evaluator

The CB for the digital circuit simulator consists of behavioral specifications for components such as AND, NAND, NOT, OR, and NOR gates. The behavioral specification for digital gates is accomplished using truth tables and gate delays. Each gate component in the CB has a record which includes:

- gate type
- number of input pins
- number of output pins
- gate delay
- output values list

The digital circuit simulator uses tri-state logic corresponding to FALSE (0), TRUE (1), and UNKNOWN (2). Therefore, if a gate has 2 input pins, its associated truth table has nine (3**2) entries. In order to reduce the amount of storage required for truth tables, only the output values are entered.

The Evaluator is responsible for determining the appropriate behavioral actions for a specific gate type. The Evaluator is passed the gate type and input net values. The appropriate output value is retrieved by indexing into the output values list using the input net values. Thus, the component's behavior can be determined and the time at which it responds is calculated as the current time plus any associated gate delay. The Evaluator returns the new output net values and the time at which the transition occurs.

3.3 Model Base

The MB for the digital circuit simulator consists of a collection of text files containing model descriptions. In the C++ implementation, OrCad is used to define the models using TTL gates. OrCad is a graphical schematic capture tool. The OrCad files are loaded into the Active Model Graph using a conversion utility provided in the digital circuit simulator.

In the C implementation, a different approach is used to create and store models. A model is specified textually using a mini-language as shown in Figure 4. This "program" is then input into the simulator which creates the model in the Active Graph Area and evaluates it when so instructed.

3.4 Simulation Manager

The simulation manager for the digital circuit simulator provides event list manipulation routines, a controller that coordinates the simulation, and a user interface.

The event list is a linked list which contains net names, transition values, and the times at which the transitions are to occur. It may be constructed as a time wheel or simply an ordered linked list. Its structure is important only for the sake of efficiency.

For each clock time, the simulation manager examines each event notice and the associated net values are updated. Each gate which is driven by, or has its input value determined by, a net which has had a value changed must be reevaluated. In turn, this reevaluation generates new event notices for all output nets of these gates.

```
* Define the original model
#DEFINE_MODEL;
N NODE001     AND;
N NODE002     AND;
N NODE003     NAND;
I  ARC001     NODE001 = TRUE;
I  ARC002     NODE001 = FALSE;
I  ARC003     NODE002 = UNKNOWN;
I  ARC004     NODE002 = FALSE;
O ARC005 [NODE001]          NODE003;
O ARC006 [NODE002]          NODE003;
O ARC007 [NODE003];
#EVALUATE_MODEL;
#DUMP_MODEL;
*  Specify changes which occur at some future point in time
*  and reevaluate
I  ARC003     NODE002 = TRUE;
#EVALUATE_MODEL;
#DUMP_MODEL;
#EXIT;
```

Figure 4. Sample Model Description for the C Implementation

The user interface for the digital circuit simulator varies for each implementation. In the C++ system, the interface is menu-driven. OrCad is used to create a graphical description of the model and a conversion utility takes the OrCad output and creates an Active Model Graph. The menu is used to specify the circuit to be loaded and to define the input stimuli. The user may run the simulation or step through it interactively. Output net values are displayed textually.

In the C implementation, a model is specified textually using a mini-language as shown above. This program provides input for the simulator. Note that as it exists currently, the C implementation does have a controller. It is up to the modeler to specify each evaluation which must take place. Because of its recent completion, the implications of this interface have not been fully investigated.

3.5 Discussion

The digital circuit simulator described above has been a very successful project. The original implementation is in Lisp on a TI Explorer [Benavides and Wyatt 1988]. The C++ implementation is under Unix on a VAX 11/780 [Wyatt and Ali 1990]. The C implementation is on an IBM PC in Turbo C and on a VAX VMS-based system. In each case, the system architecture remained relatively unchanged.

The structure of a digital circuit simulator lends itself very naturally to the GBM approach. It uniquely illustrates the concept of developing more complex models from basic building blocks. Power and flexibility for a digital circuit simulator in this framework are achieved by adding more components to the CB. In addition, verification of the digital circuits simulated using the GBM system is relatively simple because of the non-stochastic nature of the problem. Therefore, the validation of the GBM architecture was accomplished with minimal difficulties.

4. RESEARCH IMPLICATIONS

The GBM framework outlined above has been implemented in a variety of languages. Each implementation utilized similar architectures, even though the paradigm of each language was substantially different. This illustrates the flexibility of the framework. Not only is it adaptable to a variety of language paradigms, it also provides an excellent architectural basis for a variety of research problems.

4.1 Reusability Studies

The GBM framework provides an excellent testbed in which studies examining the productivity benefits of reuse may be performed. Although simulationists are not as interested in productivity studies as are software engineers (as demonstrated by the lack of published research in the simulation community), this framework readily supports studies which might be performed. It would be interesting to determine the relative productivity of a modeler developing code independently versus one functioning in this framework.

4.2 Behavorial Specification Techniques

The investigation of behavioral specification techniques for model components is a very interesting extension of this research environment. Admittedly, digital circuit components have relatively simple behavioral specifications. Truth table evaluations generally occur via pattern matching. Components which represent more complex servers such as routers and machines with breakdowns have more complicated behavioral specifications than have currently been demonstrated. However, it is believed that this framework is designed such that behavioral specification techniques may be investigated with relative ease. After defining a particular behavorial specification technique suitable for the application domain, the structure of components in the component base must be modified and an evaluator which interprets the specification must be written. The clean separation of subsystems makes this task less painful than it might otherwise be. Plans exist for developing more complex component behaviors in other application domains in the near future.

4.3 Graphical Interfaces

This framework for graph-based models provides an excellent setting for the investigation of graphical interfaces strategies for defining simulation models. It provides an environment in which techniques for specificating behavior can be combined with their graphical interfaces to determine which provides the modeler with an friendly and flexible conversation mechanism. The use of graphical output may be investigated with similar efficiency.

4.4 Embedded Expert Systems

Finally, this framework supports the inclusion of embedded expert systems as defined by O'Keefe [O'Keefe 1986]. The original Lisp implementation of the digital circuit simulator has an expert system embedded in it which monitors the event list and Active Model Graph area to determine if a behavioral phenomena condition might occur. Although missing on the current versions of the digital circuit simulator, it is scheduled for inclusion. This inclusion will open the doors for further research on how AI reasoning can assist simulators and on how this knowledge might be shared between models.

5. SUMMARY

A framework has been proposed in this paper for graph-based models which supports the reuse of simulation model components. Because of its utility for all problems in which graph-based models are appropriate, the potential application domains for GBM are widespread. By allowing the modeler to define a reusable library containing behavioral characteristics of model components, this system has a much greater level of flexibility over similar commercial packages which provide the modeler with a standard set of components that are tailorable at best.

In addition, the GBM architecture provides an excellent framework for a variety of research problems which are of ineterst. Behavorial specification techniques, graphical interfaces, and AI integration are interesting and useful research problems. Often, in a academic research environment, it is difficult to investigate specific problems without developing a specialized simulation shell, or prototype, which includes only minimally acceptable features of a simulation environment. This framework attempts to overcome this problem by providing a small architecture which readily supports the modeling of graph-based systems.

ACKNOWLEDGEMENTS

This work was supported in part with Faculty Research Opportunity Grants from the University of North Texas and with ACM SIGDA / IEEE DATC scholarship grants. I would like to acknowledge the contributions of John Benavides for his design of the original knowledge-based digital circuit simulator (SWIM), Blair Copeland for his work on the behavorial specifications of digital components, George Dupre for his work on the C implementation, and Bill Turney for his endless contributions and relentless work on many versions of the system reported herein. In addition, I am grateful to Dr. Mahir Ali for his efforts in procuring funds from ACM/IEEE.

REFERENCES

Ali, M. and D.L. Wyatt (1990), "An Object-Oriented Approach to Knowledge-Based Digital Circuit Simulation," In *Proceedings of the 1990 SCS Conference on Object-Oriented Simulation*, A. Guasch, Ed. SCS, San Diego, CA, 38-42.

Balci, O. (1989), "How to Assess the Acceptability and Credibility of Simulation Results," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE, Piscataway, NJ, 62-71.

Balci, O. and R.E. Nance (1987), "Simulation Support: Prototyping the Automation-Based Paradigm." In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 495-502.

Balmer, D.W. (1987), "Modelling Styles and Their Support in the CASM Environment," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ 478-485.

Benavides, J. and D.L. Wyatt (1988), "Improving Digital Circuit Simulation: A Knowledge-Based Approach," In *Proceedings of the 1988 Winter Simulation Conference*, M.A. Abrams, P.C. Haigh, and J. Comfort, Eds. IEEE, Piscataway, NJ, 362-371.

Copeland, M.B. and D.L. Wyatt (1990), "Behavorial Specification Techniques for Digital Circuit Simulation," Technical Report, Department of Computer Science, University of North Texas, Denton, TX.

Derrick, E.J., O. Balci, and R.E. Nance (1989), "A Comparison of Selected Conceptual Frameworks for Simulation Modeling," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE, Piscataway, NJ, 711-718.

Kaplan, D.J. (1987), "The Process Graph Method: An Iconic Method of Controlling Networks of Processors," In *Proceedings of the 1987 Summer Computer Simulation Conference*, J. Clema, Ed. SCS, San Diego, CA, 219-227.

O'Keefe, R. (1986), "Simulation and expert systems - A taxonomy and some examples," *Simulation 46*, 1, 10-16.

Pritsker, A.A.B. (1989), "Why Simulation Works," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE, Piscataway, NJ, 1-8.

Reese, R. and S. Sheppard (1983) , "A Software Development Environment for Simulation Programming," In *Proceedings of the 1983 Winter Simulation Conference*, S. Roberts, J. Banks, and B. Schmeiser, Eds. IEEE, Piscataway, NJ, 419-426.

Reese, R. and D.L. Wyatt (1987), "Software Reuse and Simulation," In *Proceedings of the 1987 Winter Simulation Conference*, A. Thesen, H. Grant, and W.D. Kelton, Eds. IEEE, Piscataway, NJ, 185-192.

Thesen, A. and L.E. Travis (1989), "Simulation For Decision Making: An Introduction," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE, Piscataway, NJ, 9-18.