Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

Automatic development of parallel simulation models in ADA

Carolyn K. Davis
Data Systems Division
General Dynamics
Fort Worth, TX

Sallie V. Sheppard
Associate Provost
Texas A&M University
College Station, TX

William M. Lively, Director
Laboratory for Software Research
Texas A&M University
College Station, TX

ABSTRACT

This paper introduces MultiSim, a prototype, user-oriented tool specifically designed to automate the model development process for parallel simulation models. Targeted toward the simulationist and written in Ada for high transportability among different numbers of processors, MultiSim combines discrete-event simulation knowledge, parallel programming knowledge, and target language knowledge and represents this knowledge in frame-like constructs. Through user interaction, knowledge of the system to be modeled is abstracted and a parallel Ada simulation model is automatically generated based on the knowledge resident within MultiSim.

1. INTRODUCTION

To take advantage of parallel processing in simulation applications, the simulationist must have parallel programming knowledge in order to decompose a model into a collection of separate modules that can execute concurrently on multiple processors. The problems inherent in parallel processing can make this decomposition process quite complex. A tool called MultiSim has been developed which seeks to overcome this complexity by incorporating parallel programming knowledge into the development software support system. It allows faster model development because it reduces the simulation knowledge required of the user and hides system details. MultiSim enhances the simulationist's ability to generate models without requiring extensive understanding of parallel programming concepts. The following sections discuss the knowledge needed in such a tool and briefly describe how this knowledge is maintained within MultiSim. A complete description of MultiSim, including a discussion of implementation issues and preliminary experiences in using the system, is given in Davis (1988).

2. REQUIRED KNOWLEDGE FOR PARALLEL PROGRAMMING

To support parallel model development without requiring extensive parallel programming knowledge on the part of the simulationist, MultiSim follows an approach developed by Murray and Sheppard (1988). Four types of knowledge needed to construct parallel models are included in MultiSim: *domain, simulation, parallel programming,* and *target language*.

2.1 Domain Knowledge

Domain knowledge encompasses a particular application area. A queueing network system [Banks and Carson, 1984] was selected as the domain for MultiSim. Knowledge of these systems can be described by their arrival population, arrival processes, the discipline and configuration of their queues, and their service mechanisms. There may be multiple customer types, multiple servers and multiple types of service arranged in series or in parallel. Branching customer paths and internal feed-back loops are common in the system. Thus, queueing systems provide a realistic and useful domain for inclusion in MultiSim.

2.2 Modeling Knowledge

Modeling knowledge must be included in MultiSim to allow description of the general requirements of a model. This type of knowledge includes incorporating a modeling methodology (i.e. world view) into the simulation model. Three different world views can be modeled: activity-oriented, event-oriented, and process-oriented [Banks and Carson, 1984]. Since parallelization of a model is very often achieved by decomposing the model into processes sharing entities, the process-oriented modeling approach best matches this decomposition scheme. The process view is a simple and natural way to define discrete-event simulation models and was, therefore, selected for MultiSim.

General requirements necessary for implementing the process-oriented view include characterization of entities and their flow through the system. Two types of entities are present in a simulation model: *temporary entities* and *permanent entities*. *Temporary entities* are objects which are created for a certain span of time and are then destroyed. These entities flow through the system and have actions performed on them. *Permanent entities* are objects which once created remain for the duration of the simulation. More commonly known as resources, these objects often service the temporary entities.

Temporary entity flow information must be provided to show the steps a temporary entity will take through the system. It is the description and utilization of this entity flow information which is at the heart of simulation model development.

2.3 Target Language Knowledge

Target language knowledge includes an understanding of the semantics of the target language to allow the proper selection of constructs for model implementation. Knowledge of the language syntax is also necessary to produce an executable model.

Since one of the goals of MultiSim was portability of parallel models, Ada was chosen as the target language. Unlike other languages, Ada is designed to allow true portability among architectures with differing numbers of processors. Ada also offers concurrency at the source level with the tasking construct and supports process-to-process communication with the rendezvous mechanism. The use of Ada precludes the need for load-balancing capabilities and scheduling strategies within the simulation support environment as these are included in the Ada parallel runtime system.

2.4 Parallel Programming Knowledge

Knowledge of parallel programming must also be present to support parallel model development. This knowledge involves two categories: model *partitioning* into processes and *communication* among these processes [Babb, 1988]. *Partitioning* involves dividing the model into modules that can execute in parallel. *Communication* among these modules may be necessary to continue progress in the simulation. A goal in partitioning is to divide the model so as to reduce communication between processes by pinpointing related sections of code which can reside in one module. Because delineations might not be clear cut, a flow graph representing data dependencies in the model can be constructed and rearranged to form a newer, possibly less complicated version with fewer dependencies. Once these dependencies are understood the model can be partitioned to reduce communication among modules.

Communication flow is not the only criterion on which partitioning can be based. *Granularity* of the module can also be considered. *Granularity* is the level of parallelism within a module [Babb, 1988]. Coarse granularity allows more computation to be performed in a single module thus requiring less frequent interaction between the modules. This is manifested in less communication. Unfortunately, coarse granularity may not exploit all the parallelism in a model since the activities within a single module will be executed sequentially. Fine granularity, in contrast, reduces the computation performed within each module, thereby enhancing potential parallel execution of the modules. However, less processing in a module may necessitate more frequent communication with other modules. Thus, the cost of communication is a significant factor when partitioning the models. MultiSim utilizes fine granularity since this level of granularity provides flexibility by allowing MultiSim to eventually apply any level of granularity when configuring the model for parallel execution.

Once partitioning is complete a communication mechanism must be selected whereby modules will interact. The target architecture is usually the main consideration for this decision and can be distributed (loosely coupled) or concurrent (tightly coupled). A distributed simulation scheme has been incorporated into MultiSim. This scheme contributes to MultiSim's portability since distributed simulation will execute on both loosely-coupled and tightly-coupled architectures.

These four types of knowledge -- *domain, modeling, target language* and *parallel programming* -- play an integral role in the development of parallel simulation models. Figure 1 illustrates the use of these knowledge bases in the development process. Domain and modeling knowledge enable the user to describe the model specification. The specification is transformed into a parallel simulation model by utilizing modeling, parallel programming and target language knowledge.

The processing performed by MultiSim consists of three stages as illustrated in Figure 2. In the first stage the model specification is extracted from the user's description of the attributes of the model. This specification forms the basis for the other two stages. During the second stage the model specification is analyzed to identify and represent the entity flow patterns within the modeled system. A representation scheme is needed to assure easy and quick storage and retrieval. In the final stage the parallel simulation model is generated utilizing the knowledge obtained in stages one and two. This knowledge is transformed into executable Ada code.

The key to successful translation of a specification to an executable parallel model is representation of the flow pattern within the modeled system. A representation scheme which allows easy storage of knowledge and which can still represent the flow of the entities through the system is needed. The frame-based approach in artificial intelligence offers this advantage. Originally proposed by Minsky [Barr and Feigenbaum, 1982], frames are a method of organizing knowledge in a way that directs attention and facilitates recall and inference. Accompanying this methodology is the idea of scripts. Developed by Schank and Abelson [Barr and Feigenbaum, 1982] scripts are frame-like structures specifically designed for representing sequences of events.

Frames provide a structure within which new data are interpreted in terms of concepts acquired through previous experience. Furthermore, the organization of this knowledge facilitates expectation-driven processing, looking for things that are expected based on the context one thinks he is in. The representation mechanism that makes possible this kind of reasoning is the slot, the place where knowledge fits within the large context created by the frame.

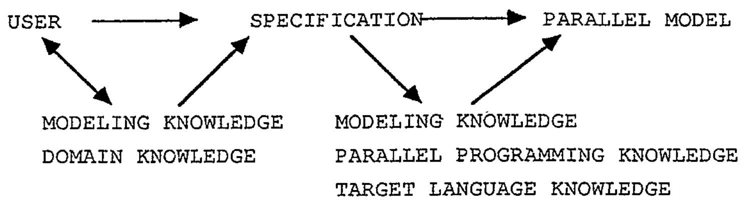

Figure 1: Use of Knowledge Bases Structures for Knowledge Representation

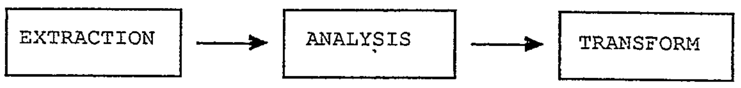

Figure 2: Stages in MultiSim

After a particular frame or script has been selected to represent the current context or situation, the primary process in a frame-based reasoning system is often just filling in the details called for by its slots. These slots can be filled in three ways: *default*, *inheritance*, and *procedural attachment*. To fill a slot by default requires that a default value be provided if no other indications of a value exist. Inheritance allows a slot to be filled by another slot to which it is related. Procedural attachment attaches procedures to slots to drive the problem-solving behavior of the system. If a value does not exist for a slot, the procedure is activated to obtain the information.

These frame and script concepts suggested the representation structures which were implemented in MultiSim. Ada supports frame-based representation by employing the record data structure. Each record contains slots to be filled by the user. Procedural attachment was achieved by encapsulating the procedures for the frame in a package.

3. CLASSES OF FRAMES

Based upon the type of objects identified in simulation models, three classes of frames were implemented in MultiSim: *entity-class*, *action-class*, and *simulation-class*. The entity-class characterizes temporary and permanent objects. The action-class describes actions performed in the model. This class is used to develop entity scripts. The simulation-class encompasses experimental characteristics such as name of the modeled system, length of simulated time, a list of resources in the system, and a list of entities in the system. This frame allows separation of experimental data from model data. A change in only this frame can affect a change in the entire simulation. For example, assigning a new value to the slot *run-time* affects a new run without modifying other frames. While there can be several permanent and temporary frames, only one simulation-class per model is allowed.

3.1 Frames in Ada

The frame concept has been implemented in MultiSim using the record construct of Ada. As depicted in Figure 3, each object is represented by a record with slots for characteristics such as distribution, type of resource, name, etc.

```
TYPE entity class (distribution: dist:=unknown)
          IS RECORD
          name : vstring;
          kind : entity-type := unknown;
          time-units : units;
          stream : integer;
          next-entity : entity-class-ptr;
          CASE distribution IS
                WHEN random => null;
                WHEN uniform => min,max : float;
                WHEN normal => nmean,stan : float;
                WHEN exponential = > emean : float;
                WHEN poisson => pmean : float;
          END CASE;
     END RECORD;
```

Figure 3. Frame in Ada

Figure 3 represents the *entity-class* frame. An *entity-type* and a *distribution* are associated with each instance of entity-class. An entity can be of kind *temporary* or *permanent* and can have a *distribution* of *poisson*, *random*, *normal*, *exponential* or *uniform*. *Entity-type*, *dist*, *vstring*, and *units* are programmer defined types. The values of *unknown* indicate this is a generic frame which can be instantiated for a specific object.

Procedural attachment is established by encapsulating the frame description and the procedures for acquiring slot values into Ada packages.

3.2 Scripts in Ada

Crucial to the design of a parallel simulation model is the analysis of data dependencies between the various processes. It is necessary to understand these interactions between the processes in order to properly establish communication links. Stage two of MultiSim performs this analysis on the model specification extracted from the user.

Upon acquiring the flow information MultiSim must store this information in a structure which can be easily manipulated. While frames were directly utilized to store knowledge in the knowledge bases, the frame and script concepts were merged in MultiSim to form the *entity flow graph* (efg) structure. Similar to frames and scripts and incorporating flowchart techniques, the efg combines the advantages of all three by actually flowcharting or graphing the script for each entity.

An efg is associated with each temporary entity type. The nodes of the graph represent actions performed on the entity. The edges represent direction of flow through the system. This diagram is the script the entity will use when traversing the system. Each node is a frame or event in the script. By analyzing the entity flow graph's nodes and edges it is possible to "know" where the entity is going and where it came from, and therefore, establish the correct communication paths between the "sender" and "receiver" processes. Figure 4 illustrates an extended entity flow graph for a TV Inspection Station scenario. Its components and organization are described below.

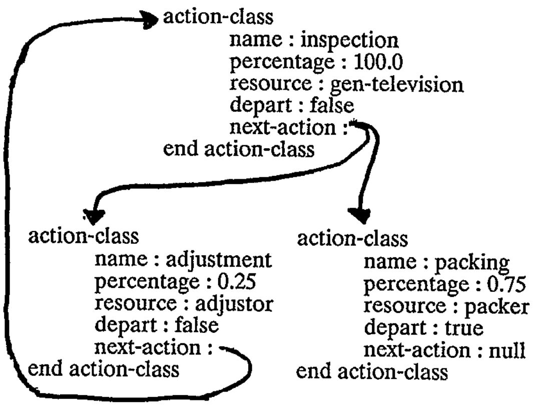

Figure 4. Entity Flow Graph for TV Inspection Station

Since scripts are basically a list of actions, an *action-class* frame was defined in Ada using the record data structure. Each instance of an action is a node of the efg structure. The edges of the efg are represented by *previous-action* and *next-action* pointers. Many instances of *action-class* are eventually linked to form the entity script.

The user describes the appropriate linking by listing in chronological order the steps an entity will follow. For example, assume three resources in a TV Inspection Station have been described as 1) inspection, 2) adjustment, and 3) packing.

These possible actions are displayed on the screen with the question, "What is the next action?" The user selects action 1, 2 or 3. This process continues until the entities depart the system. In this manner the user builds an efg structure for each type of entity in the system.

If the flow involves probabilistic branching the user specifies the number of branches to be taken and names of the branches and what percentage of entities travel that branch. Total percentage must add to 100.0. Based on this branching knowledge MultiSim elaborates each branch by utilizing a recursive depth first traversal guiding the user's input with questions specifically designed to assist the user in specifying one complete path before elaborating another path.

Once all entities have been diagnosed for entity flow, MultiSim recursively traverses the entity flow graph setting in motion the pointers, frames, and scripts necessary as input to the translator which will insert appropriate mechanisms for proper tasking.

4. TRANSLATION OF EFG TO ADA

The construction method needs to allow the transformation of the frames and entity flow graphs into target language structures. The efg is used to outline the basic structure of a model. Non-valued slots in the outline are then assigned values as available from other frames to build structures for a complete executable model. These structures declare and define permanent entities, instantiate statistics collection facilities, etc.

The frames and scripts as represented in MultiSim are easy to transform into executable Ada. The construction rules which govern this transformation are:

1. Each permanent frame is a resource.
2. Each temporary frame is an entity to be declared and created.
3. Each simulation-class frame give overall model specifications.
4. Each temporary frame and permanent frame is equivalent to a process.

To describe a process in Ada requires two parts: a *receive-entity* section and a *service-entity* section. The receive part specifies what entity is to be received and from where. The service part describes the service performed on the entity and states where the entity will go next. Once the construction stage is complete, i.e. all components of the basic model have been transformed into target language structures, these structures can be formatted and written as statements in files with the appropriate syntax and proper punctuation of the target architecture.

5. CONCLUSION

Parallel simulation requires a different approach to programming. Instead of requiring the user to learn this approach it is more practical to incorporate parallel programming techniques into the development process. MultiSim does this by incorporating four types of knowledge into the support software. Through interactive dialog, knowledge of the system is specified and stored in structures similar to frames and scripts. The vital flow information is stored in an entity flow graph. The efg is then translated into executable Ada code. MultiSim, therefore, automatically generates parallel simulation models yet requires no parallel programming knowledge on the part of the user.

MultiSim has been implemented in Ada and is executable on a Sequent Balance 8000. A prototype environment, ParSim, has also been implemented which allows the user to interface the models developed in MultiSim to a parallel execution environment. This environment integrates a time synchronization algorithm, simulation support routines and MultiSim via the ParSim command language.

A comparison of results using the sequential and parallel run time environments on the Sequent Balance 8000 were given in Sheppard, Davis and Chandra (1987). A complete scenario of parallel model development and execution using MultiSim and ParSim is given in Davis (1988).

6. REFERENCES

Babb II, R.G., Editor (1988). *Programming Parallel Processors*, Addision-Wesley Publishing Company, Inc., Reading, Massachusetts.

Banks, J., and Carson II, J.S. (1984). *Discrete-Event System Simulation*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Barr, A., and Feigenbaum, E.A., Editors (1982). Knowledge representation. In *The Handbook of Artificial Intelligence Vol.* I. William Kaufmann, Inc., Los Altos, California.

Davis (Hughes), C. K., Chandra, U., and Sheppard, S. V. (1987). Two Implementations of a Concurrent Simulation Environment. In *Proceedings of the 1987 Winter Simulation Conference* (A. Thesen, H. Grant, W. D. Kelton), 618-623.

Davis, C. K. (1988). A Generator for Parallel Simulation Models in Ada. Unpublished Ph.D. Dissertation, Computer Science Department, Texas A&M University.

Murray, K. J. and Sheppard, S. V. (1988). Knowledge-based Simulation Model Specification, *Simulation*, 50, 3, 101-111.

AUTHORS' BIOGRAPHIES

CAROLYN DAVIS received her Ph.D. in Computer Science from Texas A&M University in August 1988. She is currently employed by General Dynamics in the Data Systems Division where she is involved in research to apply the Ada programming language to real-time simulation using multiple processors. She is a member of IEEE Computer Society, ACM, and SCS.

Dr. Carolyn K. Davis
Data Systems Division
General Dynamics
Fort Worth, TX 76112
(817) 777-8483

SALLIE V. SHEPPARD is Associate Provost for Honors Programs and Undergraduate Studies and is a Professor in the Department of Computer Science at Texas A&M University. Her research interests include simulation and artificial intelligence for software engineering. She received her Ph.D. in Computer Science in 1977. Dr. Sheppard is a member of IEEE Computer, ACM, and SCS. She is on the Board of Governors for the IEEE Computer Society and is their representative to the Winter Simulation Conference Board.

Dr. Sallie V. Sheppard, Associate Provost
Honors Programs and Undergraduate Studies
103 Academic Building
Texas A&M University
College Station, TX 77843-4233
(409) 845-3210

WILLIAM M. LIVELY is Director of the Laboratory for Software Research and Associate Professor in the Department of Computer Science at Texas A&M University. He received his Ph.D. in Electrical Engineering and Computer Science from Southern Methodist University in 1972. His research interests relate to applying knowledge base techniques and software development environments to automate the software development process. Dr. Lively is an active member of the IEEE Computer Society and is currently a member of a SIGADA working group developing and refining the METHODMAN document.

Dr. William M. Lively
Laboratory for Software Research
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
(409) 845-5480