Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

APPLICATION OF HIERARCHICAL MODELING CONCEPTS TO A MULTI-ANALYSIS ENVIRONMENT

Joel Luna
Dynamics Research Corporation
60 Frontage Road
Andover, Massachusetts 01810

ABSTRACT

An environment developed by the author which enables the application of multiple analysis formalisms or paradigms to a commonly defined problem is extended to include hierarchical modeling concepts. A description of the development of the object-oriented environment is provided as background. Concepts of model development are introduced in order to emphasize the value of hierarchical modeling concepts. Implementation of these concepts in the environment is then discussed. The implementation of experimental frame concepts, in which system behavior represented in the model is separated from behavior related to experimentation, is explored. Hierarchical concepts are then applied to the pairing of a model with an experimental frame in order to address future simulation goals such as hybrid and multi-level simulation.

1 BACKGROUND

The effort to develop an environment which would provide for the application of several analysis methods to a single problem resulted from an in-house effort to analyze a logistics pipeline network. Various tools were used to analyze the flow of failed parts through a repair network. The first approach involved formulating the repair system as a set of differential equations and solving them analytically for measures such as mean number of failed units at each repair facility (the closed-form or CF approach). This did not allow for constraints on selected facilities or modeling the pipeline as a closed loop system (fixed population). The next approach then was to analyze the system using continuous simulation (the CS approach). One advantage of this approach was that it was much easier to model the system since the differential equations did not have to be solved analytically. However, this approach did not allow the explicit modeling of facility constraints, such as a finite number of servers per facility. In addition, various statistical measures were desired such as standard deviation. Therefore, the model was then implemented as a discrete-event simulation in Simscript II.5 (the DES approach). Out of this entire process two things became clear:

1) We were constantly redefining essentially the same model of the system (that is, a collection of interconnected states with associated data such as mean service time).

2) More than one approach was useful, especially utilizing the CF and CS approaches for quick-look results and DES for somewhat longer run times but with less limiting assumptions.

Therefore, we decided it was worthwhile to investigate an environment in which these approaches could be applied to a commonly defined problem. Such an environment was implemented using object-oriented techniques and the Smalltalk programming language and environment (Goldberg and Robson 1983, Digitalk 1988).

This initial effort was described in Luna 1990 in which the CS and DES approaches were combined in a single environment. Through a single interface, a user could define a model as a set of interconnected states, each with associated measures such as mean service time and number of servers (for the DES approach). Parameters defining the experiment, such as the number of runs (DES) or integration step size (CS) and reporting interval for the collection of results could also be entered by the user. The environment then provided the option for selecting either continuous or discrete-event simulation. The underlying object structure which supported the environment consisted of formalism-specific model components (for CS and DES) and processors defined respectively as facilities and controllers. A follow-on paper (Luna 1991) extended this approach to include the CF approach as well, which performed symbolic integration and differentiation in order to formulate and solve the differential equations based on the model network defined by the user. In addition, a first attempt was made at defining the basic object framework needed to implement other formalisms, such as analytical queueing models and Petri-nets.

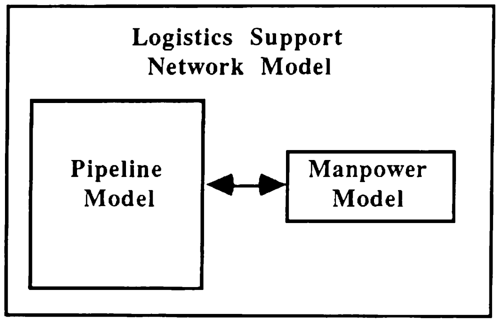
The next important step identified in the preceeding paper was to include the ability to represent any one of several possible levels of abstraction of a given model. This paper will address the issues of providing hierarchical model construction and abstraction concepts (particularly those in Zeigler 1984 and Kim and Zeigler 1987) to this environment. Applying hierarchical techniques will also lead to implementing the future requirements of simulation environments as described in Oren and Zeigler 1986 (such as hybrid analysis/simulation approaches). Changes to the original object structure required to implement a more modular approach will be described.

2 CONCEPTS OF MODEL DEVELOPMENT

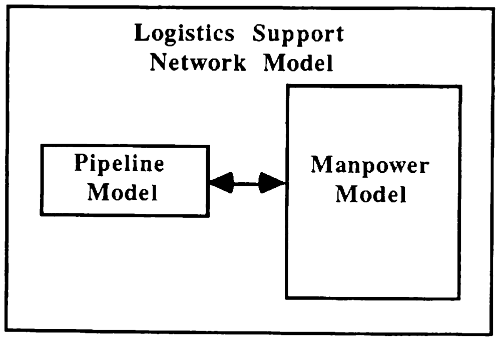
Since the environment is oriented particularly to model development, some concepts of model development merit discussion. Model development and execution takes place in the larger context of system design and analysis. Since models provide cost effective means of evaluating existing systems or system designs, they are often utilized for decisions made about the actual or intended system. Thus, the models must be able to represent the system sufficiently such that decisions made regarding the system will have the effect on the system as predicted by the models. Models may be required to represent the system at various conceptual levels. For example, in top-down system design, functions are allocated based on system requirements in successive stages from the more abstract to the more concrete. This process is generally described as concretization. At each stage, questions of optimal allocation of functions (such as tradeoff of alternative system designs) can be answered using models of the system at that stage. Thus, model development should be able to be conducted top-down to correspond with top-down system design.

On the other hand, the system may already exist but may require investigation (for example, to correct a problem). A bottom-up approach may be needed to provide a representative model of the system. In this approach, specific behavior of system elements is modeled, validated, and then succesively interfaced with other models of other elements until the system is fully modeled. In order to reduce unnecessary complexity, several model elements may be combined into one element (aggregation) or represented by a simpler type of model (abstraction).

Model development must also take into account more than one view or aspect of the same system depending on the objectives of the analyst. The same system might give rise to two or more different models in which some aspect of the system is emphasized more in one than in the others. This is particularly true of large systems involving several groups or agencies. An example is given in Figure 1 in which logistics analysts concerned with spares requirements are more interested in the details of the pipeline model than the analysts concerned with manpower of a logistics support network. This points out the need for a multi-facetted modeling methodology identified in Zeigler 1984 and Zeigler 1987. An environment for system analysis based on one or more system models should provide the user the opportunity to model selected aspects of the system which are consistent with all models of the system without having to exercise all of the models. A hierarchical approach to modeling, in which the model of a system is comprised of components which are coupled together, provides the basis of multi-facetted modeling by allowing the user to select either complex or simple models to couple together in modeling the system.



In the spare analyst's view. . .



In the personnel analyst's view

Figure 1. Multifacetted Modeling Must Examine More Than One View of the System

3 HIERARCHICAL MODELING CONCEPTS

Hierarchical modeling is made possible through modularity. If the system model can be divided up into modular components with specified input and output, then the components with compatible input/output can be connected (or coupled) together to form other components. The successive coupling of components to form components which in turn can be used to form other components is the essence of hierarchical modeling (see Figure 2).

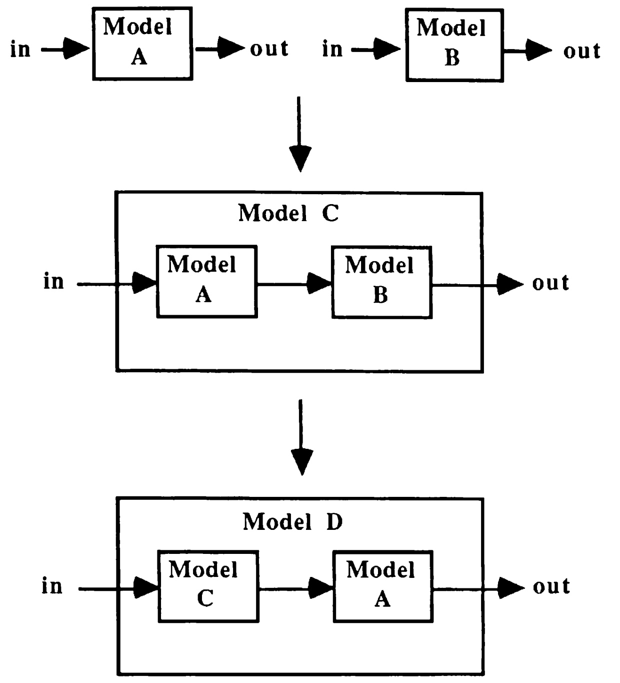


Figure 2. Successive Coupling of Modular Components is the Key to Hierarchical Modeling

3.1 Atomic Model Components

A model component falls into one of two groups. The first of these, the atomic model components, comprises the basic model building blocks from which the model is constructed. The components are considered atomic since their internal structure cannot be altered by the user. Atomic components perform the basic model behavior by their transition (external and internal) and output functions. For example, a queue as an atomic component receives input and places it in an internal collection if its server is busy, otherwise it serves the input directly. This method or procedure is an external transition function, since transition of the state of the queue is prompted by an external event. If the server becomes free and the collection of inputs is not empty, the next input is served (an internal transition function). When the server has completed servicing, it sends the served item to the next queue as its input (an output function). As a software object, the methods of an atomic component are comprised of these transition and output functions. The object's variables are those needed to perform its functions (such as the collection of inputs or the server's mean service time). Utilizing object-oriented techniques (i.e. inheritance), various types of atomic components can be defined.

3.2 Coupled Model Components

The second type of hierarchical model component is the coupled model component. These components are comprised of subcomponents (which can be either atomic or other coupled components) which are connected to each other and to the input/output interfaces of the coupled component. Input to a coupled component from an external source is passed to the 'initial' or 'first' subcomponents, and the output of a coupled component is the output of the 'final' or 'last' subcomponents which is passed to an external destination (see Figure 3). Unlike atomic components, the user may alter the structure (the number and connections of subcomponents) of the coupled component. Therefore, a coupled component must have a list of its subcomponents and their interconnections. Coupled components are used as a means of constructing and saving large scale models of systems in modular units which are more easily handled and understood. These modular units together form the model hierarchy for the user. At run time, the defined hierarchical model is implemented as a set of interacting software objects which correspond to the interconnected atomic model objects.

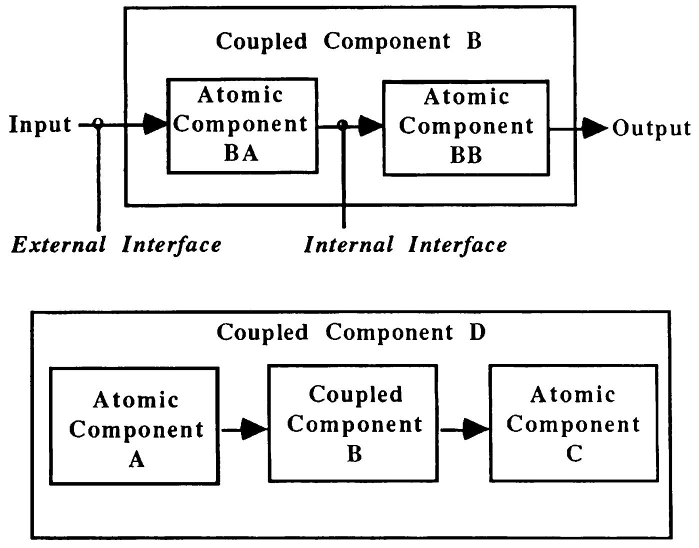


Figure 3. Subcomponents with Internal and External Interfaces for Coupled Components

4 IMPLEMENTATION OF HIERARCHICAL CONCEPTS

The impact a hierarchical approach will have on the existing environment is now ready to be examined. There is a strong correlation between the atomic components just discussed and the model objects discussed in Luna 1991. The model objects perform transition and output functions via a protocol specific to the formalism implemented. Messages utilizing this protocol are passed between the processor object and model objects, and also between the model objects themselves. Therefore, each model object must have references to the other model objects to which it is connected. Clearly, this is a coupled component feature which is embedded in the model object. Prior to a given run, the model objects and their corresponding data and connections are defined or selected. After the model objects are instantiated, they are each given the pointers or references of the other objects to which they are connected (the objects that will receive their output) as part of their initialization. The result is that the executeable objects are now coupled, and as the model is executed, the output of any model object is sent as input to those model objects which are its destinations. Since the objects each have their own coupling references, there is significantly less message traffic than if each model object had to pass all messages to a centralized message passing object which contained the coupling specifications.

In order to implement the hierarchical approach, the creation of an executeable version from a hierarchical definition of the model must be defined. First, consider the coupled components defined in Figure 3. In the previous environment, a model definition table contained a list of names of destinations for each model object as shown in Figure 4a. When the model is initialized, each of the model objects are instantiated and their reference (pointer) with their corresponding name (string) are stored by the overall environment manager (object SimMaster). Then each model object in turn obtains the corresponding reference for each of the names of destinations and stores them internally. When the model is executed, each object contains the references it needs to pass output on to its destinations. A simple modification to make in order to implement hierarchical modeling is to provide such a table for each coupled component (which contains the list of subcomponents and coupling specifications as defined before) with two additions. The first is a list of the names of the 'initial' components (those that receive input external to the coupled component). The second is a list of the names of the 'final' components (those that provide output external to the coupled component). The resulting change is shown in Figure 4b. These two lists serve as a means of providing the external coupling of a coupled component, since they indentify those components which receive input and produce output external to the coupled component. Since each coupled component is defined in the same way, these coupled components can be used to form other coupled components and thus perform hierarchical modeling. When the model comprised of atomic and coupled components is to be executed, model objects which correspond to the atomic components are instantiated as before. References to destination objects are provided as before as well, only now destinations external to a coupled component which are themselves coupled components must be found. Since the input list of a coupled component provides the list of names of subcomponents which receive input, the corresponding model objects provide the references of destination objects for those components that are sources of the coupled component. Likewise, the model objects corresponding to the list of output names provide the references of source objects for those components which are destinations of the coupled component. For example, if some component has coupled component D in Figure 3 as a destination, then the input list of component D as shown in Figure 4b identifies atomic component A as the destination object (since it receives input for coupled component D). A coupled component is not instantiated as an object per se, rather, the model objects corresponding to its atomic components are instantiated. The atomic components of successive levels of coupled components are instantiated until the resulting executeable version of the hierarchical model consists of a set of interconnected software objects which map to the atomic components.

Component	Destination
A	BA
BA	BB
BB	C
C	--

(a)

			Connections	
Component	Input	Output	Component	Destination
D	A	C	A	B
			B	C
B	BA	BB	BA	BB

(b)

Figure 4. Tables of Model Connections for a) Original Environment; b) Modified to Incorporate Hierarchical Modeling

It is important to note that aggregation or concretization can easily be implemented using hierarchical constructs. Aggregation, that is the lumping together of several components into one component, can easily be performed by replacing a coupled component at any level by a simpler coupled component or even an atomic component. Concretization can be performed by replacing atomic components with more complex coupled components. This clearly supports both top-down and bottom-up modeling approaches in support of system design and analysis.

5 TOOLS FOR HIERARCHICAL MODEL DEVELOPMENT

The implementation of hierarchical modeling requires support, particularly in the storage of atomic components, coupled components, models and corresponding data. Zeigler proposes the concept of a model base which functions as a database for model components (Zeigler 1984, Rozenblit and Zeigler 1985). When developing a model, a user should have access not only to the basic model building blocks (the atomic components), but also the pre-constructed blocks (coupled components). Therefore, these all need to be stored and made available to the user. The data corresponding to model components which need to be stored as well should be stored independently. This allows for multiple sets of data for the same component, which is necessary for performing sensitivity analyses or modeling different uses of the same component. One example of the latter is in logistics pipeline analysis in which several different parts (and thus different data) can be modeled by the same pipeline components. Once a hierarchical model has been defined by the user, possibly using model components from the model base, corresponding data must be selected from the database which is used together with the hierarchical model to form the executeable model. It appears from this that a third database would be very useful - a database which contains the hierarchical model merged with data to form the full model specification prior to execution. That way the model can be executed more than once without having to be specified more than once. More importantly, the model can be tailored directly to form new models as a form of incremental programming. Thus, three databases are required: a model base, for atomic and coupled components; a database for corresponding sets of data; and a model/database, which holds a complete model with corresponding data.

6 EXPERIMENTAL FRAME CONCEPTS AND IMPLEMENTATION

At this point, an environment in which the user is able to define a model structure, combine it with data and create executeable software objects which implement the model behavior has been described. Without input or control, however, the model objects will do nothing, and without obtaining measures no results can be provided. Thus the next step is to define the supporting environment in which the model executes. Zeigler proposes the concept of an experimental frame which characterizes the circumstances under which a model or its real system counterpart are subjected to experimentation. The principle of separating the model from the experimental frame is defined by Zeigler in part as "any data gathering/reduction (statistics, performance measurement, etc) or behavioral control (initialization, termination, etc) that is conceptually not carried out in the real system should not be placed in its model but rather formulated as part of the experimental frame". Zeigler's experimental frame has several elements: an input generator (which generates input to the model) or an input acceptor (which accepts selected parts of existing input); a run control acceptor (which checks for out of limit conditions for termination); and transducers (which provide summary mappings of model variables to measures of interest). These concepts were used as the basis for the definition of an experimental frame for the subject environment which provides input, run control, and data collection and reporting. In the previous approach, the model controller generated the input and controlled the sequence of events (including data reporting) while both the model objects and the overall environment manager object (SimMaster) utilized objects which perform statistical analysis. In particular, the model object handled its own reporting and data processing. In the experimental frame approach, these functions are allocated from model objects to new objects in order to separate model behavior from experiment behavior.

6.1 Input Generation and Run Control

The experimental frame concept is comprised of an overall Frame object and objects which fall into one of two functional parts as shown in Figure 5. The first part is the combination of the functions of input generation and run control into a generator/controller object, which with a processor object provides the mechanisms for executing the model. Since the control protocol is formalism specific, then the generator/controller object is dependent on the formalism of the atomic objects. The generator/controller would be implemented similarly to the previous model controller objects. For example, input to the CS model (for an open system) is in the form of an input rate value which is passed to the initial model objects as an argument to the message to update rate values. It should be noted that input generation is closely tied to run control for CS since the message to update rate values is one of a sequence of messages sent by the controller in each time step. The same is also true for the CF approach, since the mean state value equations are resolved for each new external input rate value. Input generation and run control are much less intertwined in DES since run control is distributed among the model objects (as events are generated) rather than centralized as it is for CS and CF. Input generation can also differ by mode, depending on whether the system is modeled as open or closed. The input for an open system is generated at various intervals, which can be deterministic or stochastic. The input for a closed system is generated and passed to the specified model objects prior to the start of the simulation. The different types of generator/controllers corresponding to the different formalisms and modes all implement the same protocol. The generator/controller class is instantiated with corresponding parameters, such as the class of input objects (for DES), interval time (for open systems), etc. The processor object provides the underlying event scheduling mechanisms which implements the time-based or event-based execution of the model objects.

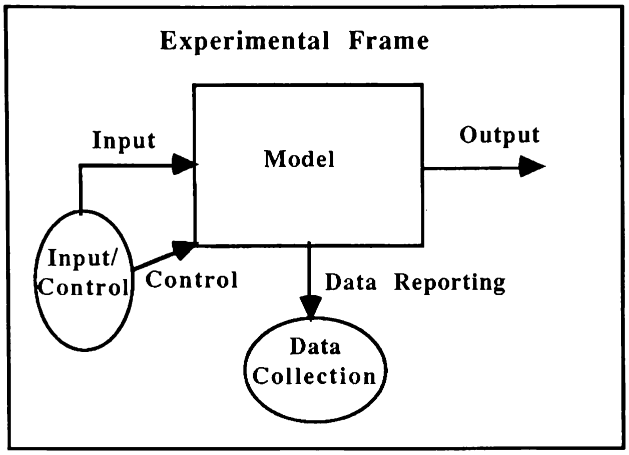


Figure 5. Experimental Frame Concept

6.2 Data Collection and Reporting

The second part of the frame concept to be implemented as an object is the data collection and reporting function. This function is performed by a group of software objects which calculate measures based on events occurring in model objects and record and/or process the values of those measures. The model objects of the previous environment were modified to eliminate the built-in computation of measures and in their place a more flexible approach was devised. Each model object transitions through a series of states (changes in status) in response to events. The measures of a model object are related to these events or changes of status. For example, the delay time in a queue is the time between a pair of events (arrival and departure of a customer). Thus, if each model object reports such events or changes in status, a separate object which receives these reports can calculate the delay measure for selected pairs of events. These objects are the data collection objects, called Probes, which are the parallel to Zeigler's transducers. Different types of probes are needed for different types of computations, thus there are a variety of Probe classes available to the user. In fact, the user may define his own measures simply by selecting a Probe class and identifying which model events or changes in status trigger the measure computation.

Since there can be more than one measure obtained for a model object, there are more than one probe and pairs of events to which probes respond. Thus, the events for each probe must be filtered from all the events being reported. The object which performs this task is the ProbeEventHandler, which provides a mapping of model events to probes. When the user defines which measures are to be obtained from which models, the ProbeEventHandler builds a table which maps the event pairs of one or more model objects to a specific probe.

Not only the values of measures are desired, but often statistics of the measures are desired as well. Another object class, StatisticsObject, and its subclasses implement statistical routines (e.g. mean, variation, standard deviation) on the measure data. Each StatisticsObject is paired with a probe so that each new measure value updates the selected statistics. More than one probe may be assigned to a single StatisiticsObject, for example, when the average of delay for several queues is desired.

The probe concept consisting of ProbeEventHandler, Probes and StatisticsObjects is shown in Figure 6. This concept can also be applied to coupled components, thus becoming hierarchical. The concept is the same for coupled components as for atomic components, only the subcomponents whose events trigger the probe must be identified as well. For example, in a coupled component of queues in series, an arrival event at the first queue is an arrival event for the coupled component, and a departure event at the last queue is a departure event for the coupled component. The delay between these two events (arrival at the first queue and departure from the last queue) is the measure of delay for the coupled component. Since the ProbeEventHandler maps atomic component events to probes, the first event is mapped with the first queue to the corresponding probe and the last event is mapped with the last queue to the same probe. The only difference for atomic components is that the queue for both first and last events is the same. Since probes can be assigned to coupled components, measures can be specified at any level. In our series of queues example, delay can be measured for each queue as well as for the series of queues as a whole. This is particularly important when trying to isolate certain types of system or model behavior, such as during debugging. Thus this probe concept provides a consistent and systematic approach to collecting data from the model. Probe structures (the assignment of probes to model components) can be pre-defined for specific measures, enabling the user to select certain measures to be obtained without having to specify the corresponding probe structure. The resulting probe structure is then objectives-driven, since it is defined based on the analysis objectives of the user. Providing both objectives-driven and user-specified probes provides the user with a great deal of flexibility in obtaining desired results.

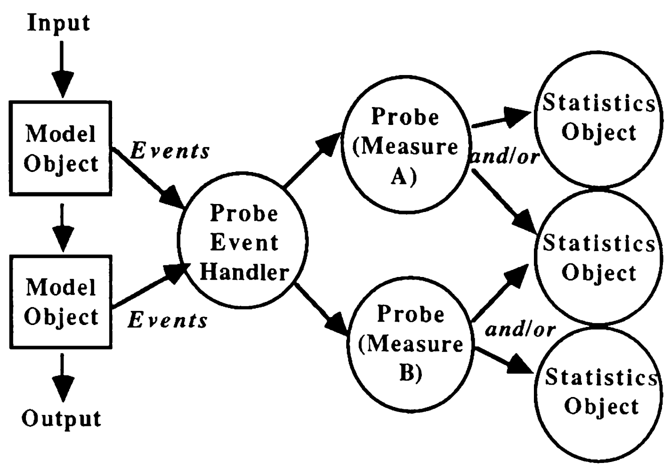


Figure 6. The Probe Concept Implementation

6.3 The Frame Object

The overall Frame object provides the instantiation, execution and data reporting of a specified model. Receiving the model, input, execution and probe definitions, the Frame object creates the model objects, interconnects them, creates a processor (if running independently), a generator/controller object, a ProbeEventHandler, probes, statistics objects, etc. The Frame object has an established protocol for model activation, suspension or termination, so that the user or even another model can execute a model via this protocol. The total frame implementation is shown in Figure 7.

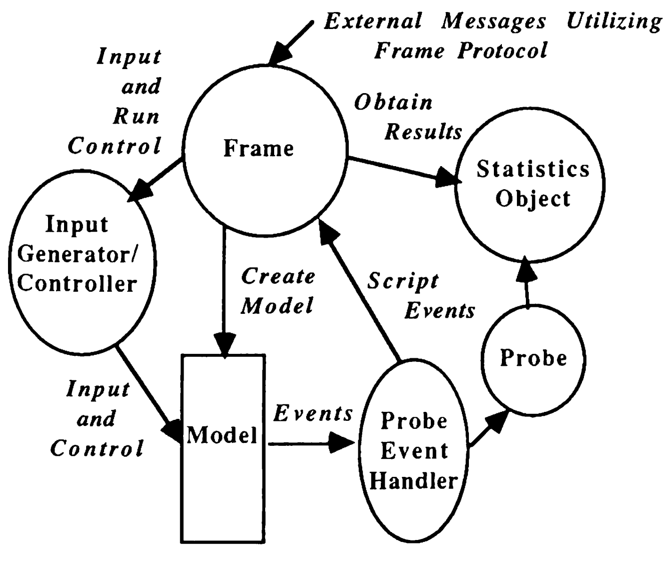


Figure 7. Experimental Frame Implementation

7 HIERARCHICAL MODEL/FRAME PAIRS

Since there is only one type of Frame object (it does not vary by formalism), it has a specified protocol which will be the same irrespective of the model it implements. Any object interacting with the Frame object will interact in the same way, making any executeable model modular through its experimental frame. Therefore, the model and its frame can be embedded in any component, provided that component implements the Frame protocol (as shown in Figure 8). The implementation of multiple levels of simulation would then be possible, such as in simulation of hierarchically composite items. For example, a unit to be repaired in a logistics pipeline may itself have subunits which may also need to be repaired. The pipelines of these subunits may themselves be dependent or independent simulations which are defined by the user and embedded in higher level model components.

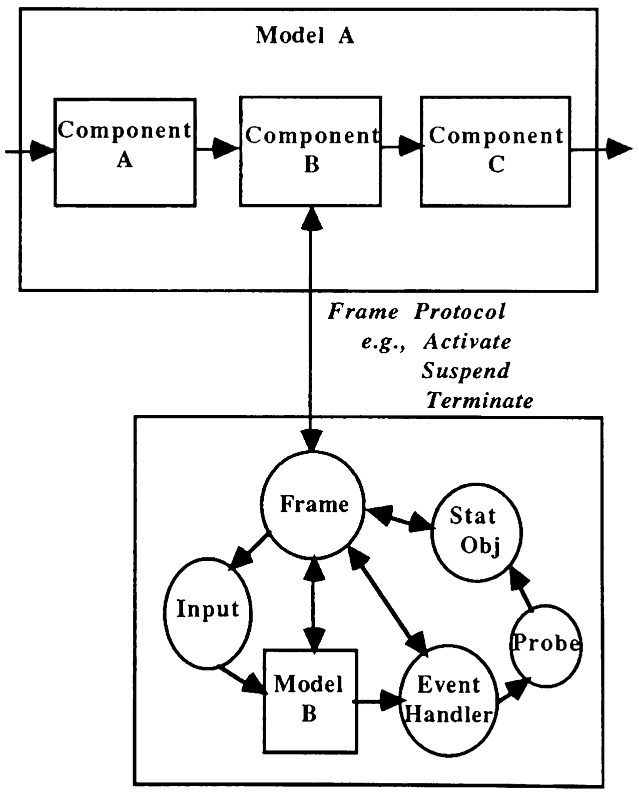


Figure 8. Embedding a Frame/Model within a Component of another Model

Since a Frame object can execute any defined model, the model embedded in another model component can be switched out, even changing formalisms. This allows the user to implement hybrid models, such as embedding a CS model in a DES model. For example, if the server of a queue can best be represented by a continuous process, a continuous simulation of the server process can be embedded in a queue which is itself part of a discrete-event simulation. This provides the user with maximum flexibility for developing and executing system models.

8 SUMMARY

The previous effort in developing a multi-analysis/formalism environment in Smalltalk was extended and modified to provide hierarchical model definition for model objects of the same formalism. An experimental frame was proposed which provides input generation, run control, and data collection and reporting by a suite of objects created by an overall Frame object. A modular model/frame approach was developed in order to implement hierarchical and hybrid simulation approaches.

REFERENCES

Digitalk, Inc. 1988. *SmallTalk/V286 Tutorial and Programming Handbook.* Digitalk Inc., Los Angeles, CA.

Goldberg, A. and D. Robson. 1983. *SmallTalk-80: The Language and Its Implementation.* Addison-Wesley Publishing Company, Reading, MA.

Kim, T.G. and B.P. Zeigler. 1987. The DEVS Formalism: Hierarchical, Modular Systems Specification in an Object-Oriented Framework. In *Proceedings of the 1987 Winter Simulation Conference.*

Luna, J. 1990. Object-Oriented Multi-Simulation Environment. In *Proceedings of the 1990 Summer Computer Simulation Conference* (Calgary, Alberta, Canada, July 16-18).

Luna, J. 1991. Object Framework for Application of Multiple Analysis Paradigms. *Object-Oriented Simulation 1991*, Simulation Series Volume 23, Number 3, 81-86.

Oren, T.I. and B.P. Zeigler. 1986. Multifacetted, Multiparadigm Modelling Perspectives: Tools for the 90's. In *Proceedings of the 1986 Winter Simulation Conference.*

Rozenblit, J.W. and B.P. Zeigler. 1985. Concepts for Knowledge-Based System Design Environments. In *Proceedings of the 1985 Winter Simulation Conference.*

Zeigler, B.P. 1984. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press Inc., Orlando, FL.

Zeigler, B.P. 1987. Hierarchical, Modular Discrete-Event Modelling in an Object-Oriented Environment. *Simulation*, November 1987, 219-230.

AUTHOR BIOGRAPHY

JOEL J. LUNA is a Senior Analyst at Dynamics Research Corporation. He is primarily involved in the application of systems analysis techniques, especially modeling and simulation, to a variety of projects. His current interests are in the areas of simulation, object-oriented programming, and effective use of graphical user interfaces.