Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

HIERARCHICAL, MODULAR CONCEPTS APPLIED TO AN OBJECT-ORIENTED SIMULATION MODEL DEVELOPMENT ENVIRONMENT

Joel J. Luna

Dynamics Research Corportation
60 Frontage Road
Andover, Massachusetts 01810

ABSTRACT

The application of hierarchical and modular properties to the simulation model construction process is identified as a need, particularly for modeling of large and complex systems. Informal definitions of modularity, coupling of modules and hierarchical construction are provided. The application of hierarchy and modularity to a modeling and simulation environment implemented in an object-oriented language (Smalltalk) is described. Object-oriented mechanisms are evaluated with respect to mechanisms needed for hierarchical model construction. The elements for construction, atomic and coupled components, and the model construction process are defined and their implementation described.

1 INTRODUCTION

Modeling is an essential part of any simulation effort. The degree to which the simulation results are able to characterize the system under study is directly related to the degree the simulation model characterizes the system. It is desirable to represent what is considered to be the important aspects or behavior of the system. For many systems, particularly large and complex systems, the model which characterizes the desired aspects of the system may itself be large and complex.

As a result, it is desirable to decompose the system model into a set of smaller parts which can be managed separately. There are two primary advantages to decomposing a problem into parts: first, all elements and facets of the task can be seen more clearly; second, the parts can be developed and tested incrementally. The result of model decomposition should be a set of model modules which can be executed independently or combined to execute as an integrated whole. The system model then is considered a modular model.

A second important property is that of hierarchy. If the result of combining model modules is itself a model module, then it can in turn be used to form other model modules. The recursion of such combinations, which is the process of hierarchical construction, results in a module hierarchy which as a whole represents the system model.

The properties of modularity and hierarchy have been applied in particular to simulation modeling (Zeigler 1984, Oren 1984). Environments for simulation model development implementing these properties have also been developed (DEVS-Scheme in Zeigler 1987, GEST in Oren 1984, and HIRES in Fishwick 1988). The author's own work in developing an environment implemented in the object-oriented language Smalltalk, in which multiple analysis methods (discrete-event simulation, continuous simulation, analytic solution) could be applied to a commonly defined problem (Luna 1990, 1991a), was extended to include an initial hierarchical capability (Luna 1991b). The focus of this paper is to further examine and implement hierarchical and modular properties in this environment.

2 ENVIRONMENT OBJECTS

A brief overview of the multi-analysis environment framework, in particular of the object structure which comprises it, will help to identify the role of the model components in the overall simulation process. The simulation model is constructed by the user through the selection and interconnection of a set of pre-coded software objects. The result of model construction is a model specification which is essentially a map of selected model objects. This map is used to create an executable model by creating (instantiating) and initializing the model objects and cross-referencing them according to the map at simulation runtime. The user also selects a data file or enters data for the model inputs.

An experimental frame is also specified by the user, which is comprised of simulation processor, input generator and model measurement objects. The simulation processor provides all of the discrete-event scheduling functions for the input generator and simulation model. The input generator provides the workload for the simulation model by generating input objects at a specified arrival rate. The model measurement objects, consisting of the probe event

handler, probe, and statistics objects, compute user selected measures based on reported model events which are then displayed to the user. The overall relationship of these objects is shown in Figure 1.

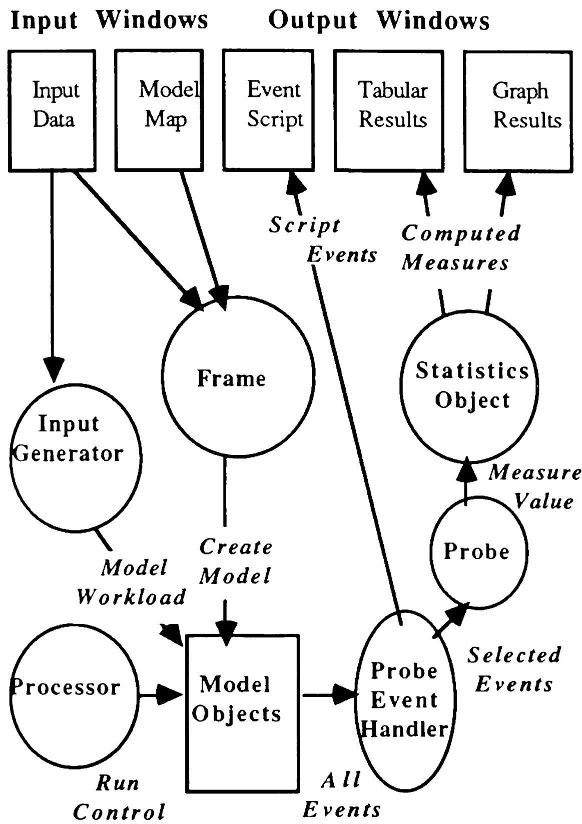

Figure 1: Environment Objects

3 MODULARITY, COUPLING, AND HIERARCHY

Before addressing the implementation of hierarchical and modular properties in the subject simulation modeling environment, a further discussion of these properties would be useful. A module is defined by Zeigler (1984) to be "a program text that can function as a self-contained autonomous unit in the following sense: Interactions of such a module with other modules can only occur through predeclared input and output ports". A module encapsulates its own internal variables and implementation details which are hidden from other modules. Access to them can only be achieved through a defined interface. In the programming language Modula-2 (Wirth, 1985), the interface of a software module is defined in its definition part, while the details are defined in its implementation part. In order to access a module's functionality, one only needs to import its definition part in order to utilize it. In order to execute the module, both parts are independently compiled and linked with other compiled modules.

Each module must have a clearly defined input/output interface by which it can be combined with other modules. For model modules, such an interface can be thought of as a collection of input and output ports. An example of this is seen in the simulation environment DeNet (Livny, 1991). In DeNet, the interface between model modules is implemented by connectors, which establish a directed interface between model objects. Through a connector one object can observe changes in the state of another object - the connector ties a set of output variables of one module to a set of input variables and events of another module.

A module composed of other modules, each of which may receive input or send output external to the module, can have multiple paths of module input and output. Each of these paths must be identified in order to unambiguously couple the modules. Assigning each path an entry or exit point to the module constitutes the definition of its input and output ports. The total set of input and output ports and the specification of what may pass through them comprises the module's interface. In Figure 2, submodules A and B have separate input so that they are assigned separate ports Ein:1 and Ein:2 (E - module E, i - input port, 1,2 - port number). On the other hand, if the user wishes to combine the output of submodules C and D, then they are both assigned to Eout:1. The process of combining or interfacing modules, defined as coupling, is the mapping of input and output ports between modules. This is also shown in Figure 2. The resulting module can itself be used to form other coupled modules. This can be seen in Figure 2 simply by substituting a coupled module E for submodule C or D since they have an identical interface (2 input ports, 1 output port). This assumes of course that the output of modules C and D are compatible with the input of modules A and B. The issue of port compatibility will be addressed later.

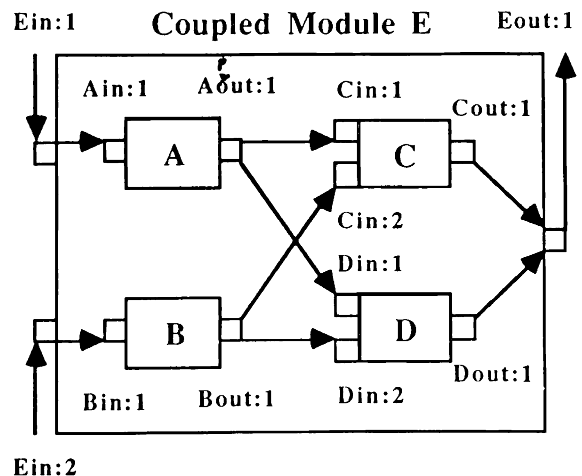

Figure 2: Connecting Modules to Form a Coupled Module

4 IMPLEMENTATION

4.1 Object-oriented Features

The language in which the author's simulation environment was developed is Smalltalk (Digitalk 1988, Goldberg and Robson 1983) which already has mechanisms useful for implementing simulation model hierarchy and modularity. Smalltalk is based on the class concept originating in SIMULA (Birtwistle 1979). The object class in Smalltalk is used to specify the variable names and procedures (called methods) each object has in creation (instantiation). Each created object or instance maintains its own variables which can only be manipulated by the object's own methods. An object executes a method when it receives a message of the same name from some other sending object. Smalltalk objects are modular in that the only interaction each object can have with other objects is through a predeclared set of messages, or protocol. Each object variable can only be accessed, if at all, through object protocol, which persists as long as the object.

Object classes are organized in a hierarchical structure. Classes are related in the hierarchy by inheritance, in which related classes share variable names and methods. The variable names and methods of a class higher in the hierarchy are a subset of those in a class lower in the hierarchy. The class higher in the hierarchy is defined as the superclass to classes lower than in it in the hierarchy, which are defined as subclasses. Thus, a subclass has access to all the variable names and methods of the superclass, and has additional variable names and methods or both itself. In this way the subclass specializes the behavior of the superclass. While this provides a hierarchical specialization of objects, the structure for hierarchical decomposition is not explicitly provided.

The interface of Smalltalk objects is by message passing from source (the sender) to the receiver. In fact, the statement syntax of Smalltalk is in the form

receiver message(: argument - optional).

where receiver is the name of the receiver object, message is the name of the method the receiver is to implement, and an argument may be supplied with the message. The source object is implied since it is the object in which the statement is implemented. The receiver is identified as a variable in the sender's code (either directly labeled or indirectly as a member of a labeled set) which holds the receiver's reference or address at run time. The full set of messages sent by one object to another is the protocol directed from the sender to the receiver. This will become important in ensuring valid couplings during the hierarchical construction process.

4.2 Atomic and Coupled Modules

This section will address the implementation of hierarchical and modular simulation model modules using the Smalltalk object-oriented mechanisms. It is helpful first to reconsider the model decomposition process. The system model is decomposed into successively smaller problems interrelated by a hierarchical tree structure. In this structure each node represents a module, and the leaf nodes represent atomic modules - modules which are not further subdivided. An example of this structure is shown in Figure 3, in which coupled module E is substituted for atomic module C from Figure 2 to form a new coupled module F.

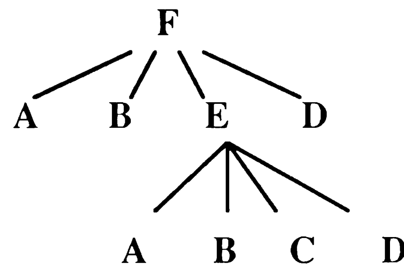

Figure 3: Hierarchical Tree of Mappings

Each leaf node is implemented in the environment as a Smalltalk class which defines the behavior of each module. The other nodes (root and branch) are results of coupling atomic and other coupled modules. Each coupled module is essentially a map of modules. Thus, the coupled module is not a software object, rather, it is a specification or map of the interconnection of its member modules or submodules. The root model, the system simulation model, is a hierarchical mapping of the underlying Smalltalk objects. Hierarchical construction then becomes the process by which these maps are defined.

The coupled module is a named specification which identifies its submodules, their type, their interconnection (internal mapping) and their connection with the module ports. An example is shown in Table 1. If a submodule is an atomic module, its type corresponds to the Smalltalk class name. If it is a coupled module, its type corresponds to the name of a coupled module. Since a coupled module does not consist of actual code, the user can create any number or type of coupled modules for use or reuse without having to do any programming.

Table 1: Coupled Module Specification

Submodules	Type
A	atomicModuleA
B	atomicModuleB
E	coupledModuleE
D	atomicModuleD

Internal Map	External Map
Aout:1 -> Ein:1	Fin:1 -> Ain:1
-> Din:1	Fin:2 -> Bin:1
Bout:1 -> Ein:1	Eout:1 -> Fout:1
-> Din:1	Dout:1 -> Fout:1

4.3 User Model Construction

The environment should support the process of model construction. In creating a coupled module, the user should be able to select a set of submodules from a module library of both atomic and coupled modules. The user should then be able to interconnect them with assistance from the environment, particularly in ensuring port compatibility through port protocol checking. The user should also be able to define a set of input and output ports to which the submodules are connected. Once named, the environment should save the new coupled module in the module library for browsing and selection.

4.4 Protocol Checking

The issue of port compatibility in a message passing environment is one of compatible protocol. Before two ports are connected, one must ensure that the source and receiver protocol are compatible in order to avoid any errors. In Smalltalk, an error is reported when a source attempts to send a message for which the receiver does not have a corresponding method. In order to ensure that the receiver will not be sent any messages it does not have, the entire set of messages which the source will send must be a subset of the set of methods which the recevier implements. If this condition exists, then the protocol will be compatible, and ports connecting these two objects will be compatible as well. Ensuring compatibility then involves checking the output protocol of the source with respect to the input protocol of the receiver.

The simplest example of protocol checking is that of a coupled module which has only one submodule (an atomic module) and only one input and one output port. In this case, the valid input port protocol is the list of methods implemented by the atomic module (a Smalltalk object). The valid ouput port protocol is the set of messages sent by the object to its receiver. If two of these modules are connected, then the Smalltalk object protocol is compared for compatibility. An example is shown in Figure 4.

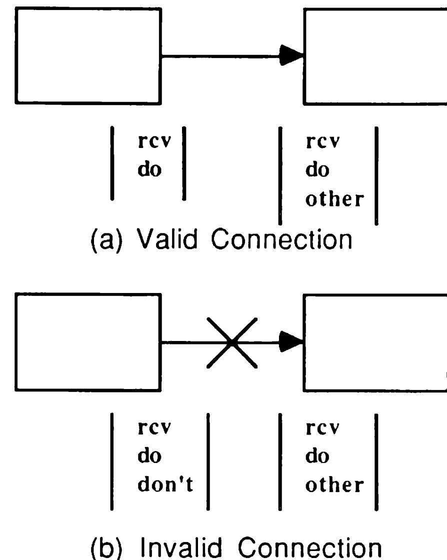

Figure 4: Protocol Checking for Valid Port Connection

If a coupled module has more than one atomic module, there may be more than one input and output port. There can be as many input ports defined as there are submodules (assuming a one-to-one correspondence of input ports to submodules) or as few as one input port (assuming a one-to-many correspondence of input port to submodules). In the former case, the port input protocol corresponds one-to-one with the submodule input protocol. In the latter case, the port input protocol corresponds to the intersection of the sets of submodule input protocol. This is because the valid input protocol of a module must be at least a subset of the input protocol of each submodule, and the only set of input protocol which is a subset of each submodule's set of methods is the intersection of all the submodules' set of methods.

For example, consider three submodules with method sets (*go*, *stop*), (*go*, *stop*, *pause*), and (*go*, *stop*, *goSlow*). If these submodules are all connected to the same input port, then they all must be able to receive the same protocol. If an external model module sends either *go* or *stop*, all three submodules can respond. If *pause* or *goSlow* is sent, only one of the three submodules can respond. The other two will report an error. The only set of protocol sent that will not yield an error is (*go*, *stop*), which is the intersection of the three submodule method sets. This set then is defined as the input port protocol for this port. Any module port connected to this port must have a protocol which is a complete subset of this protocol.

The output ports are handled similarly, only the set of output protocol for each port is the union of the sets of output protocol of each submodule to which it is connected. The union, rather than the intersection, of

the sets of protocol is appropriate because the output port protocol must contain all possible protocol which an input port of another coupled module will receive. If in our previous example the three sets were sets of messages sent to the receiver, then the receiver must have a corresponding method for each one or an error will result. The set of all possible messages is (*go, stop, pause, goSlow*), which is the union of the three sets. This set then is the output protocol for this port.

The protocol for input and output ports is saved as part of the coupled module's specification. Otherwise, searching for object protocol would become increasingly prohibitive as the coupled modules become increasingly nested. By identifying the protocol associated with each port of a coupled module as part of its specification, protocol checking can be conducted by the environment to ensure a valid connection between modules is being made.

4.5 Model Execution

When the user is ready to execute the constructed model, having already defined the measures to be obtained (which are implemented by the probe and statistics objects as described in Luna 1991), the environment builds and executes the simulation model. The environment does this by first instantiating all Smalltalk objects which correspond to atomic modules in the system model. Then starting at the root of the hierarchical tree structure, it attempts to connect submodules by pairing source and receiver objects. If the submodule is a coupled module, then its submodules connected to the port in question are also searched. Once the search has reached the leaf nodes (atomic modules), then the corresponding instantiated Smalltalk objects are used to make the connection. The receiver variables in the source objects are set with the references of the receiver objects. This is performed for each connection until the hierarchical mappings are fully implemented. The result is a set of executable, cross-referenced Smalltalk object instances which implement the simulation model. The model is executed by the environment using the objects previously described.

5 FUTURE WORK

The implementation of hierarchy and modularity in the subject simulation environment can be summarized as a hierarchy of mappings of Smalltalk objects which the environment uses to create executable models. While the method of allowing the user to construct models from pre-defined units in a building block fashion is not unique, the implementation of this approach in Smalltalk allows the user to modify and add objects for use in constructing models. There are two extensions to this approach, however, which will increase the flexibility of the user to build his own models.

The first is the translation of protocol between modules. If the set of atomic modules is allowed to be an open set to which modules can be added from external sources, then some translation of protocol identical in function would be very useful. For example, if a source object sends a message *whatIsYourStatus* to a receiver object with method *giveStatus*, an error will occur even if the function is the same as that intended by *whatIsYourStatus*. One solution is to create a Smalltalk class which acts much like the DeNet connector. The objects instantiated from this class would act as a shell around the receiver object, in which the method names would correspond to those sent by the source object and the method would consist of sending the corresponding method name in the receiver to the receiver. Thus the method for *whatIsYourStatus* would look like

```
whatIsYourStatus
    receiver giveStatus.
```

where receiver references the receiver object. Any number of these objects could be defined for any pairing of source and receiver objects from among the atomic modules.

The second extension would be the implementation of coupled modules as Smalltalk objects which the user could define within the environment. These objects would provide for variables and methods at the coupled module level, so that not only would the behavior of a coupled component be defined by the behavior of its atomic modules, but by its own methods as well. These modules should be allowed to have not only atomic modules, but coupled modules as receivers as well within the method statements. This would provide the user with greater flexibility in defining and modifying behavior at all levels of the model hierarchy.

REFERENCES

Birtwistle, G.M. 1979. *Discrete-event Modeling in SIMULA* New York: MacMillan.

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

Fishwick, P.A. 1988. The role of process abstraction in simulation. *IEEE Transactions on Systems, Man, and Cybernetics*, 18:18-39.

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

Livny, M. 1991. *DeNet: An overview*. Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin.

Luna, J. 1990. Object-oriented multi-simulation environment. In *Proceedings of the 1990 Summer Computer Simulation Conference*, ed. W. Svrcek and J. McRae, 56-61. Society for Computer Simulation, Calgary, Alberta, Canada.

Luna, J. 1991a. Object framework for application of multiple analysis paradigms. In *Object-Oriented Simulation 1991*, ed. R. K. Ege, 81-86. Society for Computer Simulation, Simulation Series Volume 23, Number 3.

Luna, J. 1991b. Application of hierarchical modeling concepts to a multi-analysis environment. In *Proceedings of the 1991 Winter Simulation Conference*, ed. G. M. Clark, W. D. Kelton, and B. L. Nelson, 1165-1172. Phoenix, Arizona.

Oren, T.I. 1984a. Model-based activities: A paradigm shift. In *Simulation and Model-Based Methodologies: An Integrative View*, ed. T.I. Oren, B.P. Zeigler, and M.S. Elzas, 3-40. Amsterdam: North-Holland.

Oren, T.I. 1984b. GEST - A modelling and simulation language based on system theoretic concepts. In *Simulation and Model-Based Methodologies: An I ntegrative View*, ed. T.I. Oren, B.P. Zeigler, and M.S. Elzas, 281-335. Amsterdam: North-Holland.

Wirth, N. 1985. *Programming in Modula-2*. 3d ed. New York: Springer-Verlag.

Zeigler, B.P. 1984. *Multifaceted modelling and discrete event simulation*. Orlando: Academic Press.

Zeigler, B.P. 1987. Hierarchical, modular discrete-event modelling in an object-oriented environment, *Simulation* 49:219-230.

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, modeling, and object-oriented programming.