Proceedings of the 1999 Winter Simulation Conference
P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.

SCALING, HIERARCHICAL MODELING, AND REUSE IN AN OBJECT-ORIENTED MODELING AND SIMULATION SYSTEM

Thorsten Daum
Robert G. Sargent

Simulation Research Group
Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY 13244, U.S.A.

ABSTRACT

Three useful modeling techniques for specifying discrete event simulation models are discussed. Hierarchical model specification provides for model specification at different levels of abstraction. Scaling of model elements provides for the combination of similarly structured and parallel operating model elements into arrays of both fixed and dynamic sizes. Reuse of model elements allows for the repeated use of model elements specifications. The Hierarchical Control Flow Graph Model paradigm is used to demonstrate the techniques discussed.

1 INTRODUCTION

This paper discusses several modeling techniques that are desirable for specifying Discrete Event Simulation (DES) models. In modeling complex systems one needs hierarchical modeling and scaling to model complex systems. Reuse of model elements (MEs) is needed to effectively model systems. A simulation system should provide for both specifying new MEs and have a library of MEs that can be reused.

We will use the Hierarchical Control Flow Graph (HCFG) Model paradigm (Fritz and Sargent 1995, Sargent 1997) to demonstrate how the discussed techniques can be used. The HCFG Model paradigm is a hierarchical model paradigm for DES that includes scaling and reuse of MEs. HCFG Models use two complimentary types of hierarchical model specification structures, one to specify components and their interconnections and the other to specify atomic component behavior. Subsection 2.1 gives a brief overview of the HCFG Model paradigm.

Subsection 2.2 discusses hierarchical model specification. Hierarchical modeling provides for the specification of large models with reasonable effort by allowing a modeler to break up a complex problem into smaller more manageable sub-parts, at arbitrary levels of detail. Also discussed is the ability to move among the different levels of a model hierarchy.

Section 3 discusses the scaling of MEs. We define scaling as the combination of (single) MEs into arrays. ME arrays can be static, i.e., of a fixed size, or dynamic, i.e. the array size can be changed at runtime. Scaling also has to address how ME arrays can be connected to maintain ME inter-operability.

Section 4 discusses the reuse of MEs. The ability to reuse MEs is crucial for effectively building non-trivial models. Reuse includes the reuse of newly created MEs as well as pre-built MEs contained in ME libraries.

2 HIERARCHICAL MODEL SPECIFICATION

In this section, the HCFG Model paradigm is briefly described. The concept of hierarchical modeling is introduced and different hierarchical modeling techniques are briefly discussed.

2.1 The HCFG Model Paradigm

HCFG Models define a modeling paradigm for DES modeling. Conceptually, HCFG Models consist of a set of independent, encapsulated, concurrently operating (Atomic) Components where each (Atomic) Component has its own thread of control and the Components interact with each other solely via message passing. Two primary objectives for HCFG Models are: (i) to facilitate model development by making it easier to develop, maintain, and reuse models and MEs and (ii) to support the flexible and efficient execution of models.

In an HCFG Model, the model Components and their interconnections (i.e., the Channels) are specified via a Hierarchical Interconnection Graph (HIG.) A HIG is a hierarchical structure which allows a modeler to specify model

Components hierarchically by supporting the concept of “coupling” together existing model Components to form new model Components. Each model has exactly one HIG.

The basic building block in the HIG is the model Component. Model Components are encapsulated entities which have an external view and an internal view. From the external view, all model Components have the following attributes: a name (instance name), a type (type name), a set of input ports, and a set of output ports. (Internal views are covered below.) The distinction between “instance” and “type” is significant. If multiple model Components are “instances” of the same type of Component, then those Components all share the same type definition.

HCFG Models use two different classes of model Components: Atomic and Coupled. An Atomic Component (AC) is an independent, encapsulated, concurrently operating entity whose behavior is specified via a corresponding Component behavior specification, which gives the AC’s internal view.

Coupled Components (CCs) are encapsulated model Components formed by coupling together other Components (atomic and/or coupled) to form new Components. CCs do not have behavior specifications. The internal view of a CC is the view from inside the Component but outside all enclosed subcomponents. The internal view of a CC is specified via a “Coupled Component Specification (CCS)”. A CCS specifies (i) a set of subcomponents which are coupled together to form a new CC type and (ii) how those subcomponents are interconnected. Note that a CCS defines a Component type and all instances of that type of Component in a model share the single type definition.

Each AC is encapsulated and has an HCFG, a set of (local) variables including a (local) simulation clock, and a point of control. The behavior of each type of AC is specified by an HCFG, which is state based. An HCFG is a hierarchical structure which allows a modeler to specify an AC’s behavior by recursive decomposition of its state space into a disjoint set of encapsulated partial behaviors called Macro Control States (MCSs) (pronounced “max” as in “maximum”). A MCS is specified via a MCS specification structure, which is an augmented directed graph where the nodes are (other) MCSs and/or control states. A control state is a formalization of the “process reactivation point” (Cota and Sargent 1992). Edges leaving MCSs in the augmented graphs have no attributes while edges leaving control states have three attributes: a condition, a priority, and an event. The condition specifies when an edge can become a candidate for traversal by the point of control, the priority is used to break ties when more then one edge is a candidate for traversal at the same simulation time, and the event is executed whenever that edge is traversed via the point of control during simulation execution. An HCFG consists of a set of MCS specification structures that are organized in a hierarchical way. An AC’s state is determined by which control state the AC’s point of control is currently located at. An AC changes its state when the AC’s point of control moves from its current control state over an edge in the AC’s HCFG to an adjacent control state. Note that all instances of the same AC type have the same HCFG specification.

Each HCFG Model has a model tree. A model tree consists of a HIG tree and a HCFG tree for each AC. The HIG tree contains the hierarchical relationships of the components where the leaf nodes are ACs, the internal nodes are the CCSs of the CCs, and the root node is the CCS of the top CC which encloses the entire HCFG Model. An HCFG tree contains the specification structures of the MCSs in an AC’s HCFG as its nodes and its root MCS encloses the entire behavior specification of that AC. In the model tree, each leaf node of the HIG tree has that AC’s HCFG tree. Thus a model tree shows the two-tiered hierarchical structure of an entire HCFG Model.

MEs in the HCFG Model paradigm include CCs, ACs, MCSs, edge conditions, and events. These MEs have type–instance relationships and share importance characateristics such as reuseability.

A prototype simulation system that implements the HCFG model paradigm has been developed called HiMASS-j, which stands for <u>Hi</u>erarchical <u>M</u>odeling <u>a</u>nd <u>S</u>imulation <u>S</u>ystem–<u>J</u>ava. It is an object-oriented software system written entirely in Java (Eckel 1998). HiMASS-j provides Visual Interactive Modeling (VIM) capabilities to build, modify, and execute HCFG models. It was used to develop the examples in this paper. An introduction to HiMASS-j can be found in Daum and Sargent (1997) and an in depth discussion of the system is given in Daum (1998).

2.2 Hierarchical Modeling

As modelers build more complex models, hierarchical modeling becomes an important issue. Hierarchical modeling provides the ability to partition a model specification into components which in turn can be recursively partitioned into (sub)components, resulting in a hierarchical specification structure of the model.

Partitioning models into hierarchical subcomponents can be crucial for the manageability of complex models. Without hierarchical specification structures, it can be a challenge for VIM systems to present large models, which may contains hundreds of Components, on a limited screen space. Hierarchical modeling allows for the specification of the model at different levels of abstraction, which can help in the verification and validation of a model.

The HCFG Model paradigm provides two complimentary hierarchical specification structures: a HIG that contains CCs and ACs and a set of HCFGs that contain MCSs.

To maintain manageability of models that may have hundreds of model elements (MEs) it is necessary to combine

MEs into groups at various levels. Coupled MEs should be arranged hierarchically so the modeler can specify the model at different levels of complexity, implementation detail can be hidden, and the model can be shown at arbitrary levels of detail or coarseness.

When building a model, a modeler should be able to easily move between the different levels. It should be possible to use different strategies to specify models, i.e., "top-down", "bottom-up", or mixed approa-ches, as is appropriate for a particular model.

According to the HCFG Model paradigm, an HCFG Model consists of a set of independent and encapsulated Components. HCFG Models use two complementary types of hierarchical model specification structures. This "layered approach" to modeling (Henriksen 1996) allows for the management of complex models and provides a wide range of support for reuse. The layered approach allows a modeler to specify models at different levels. A modeler can use MEs from a library or build MEs from scratch. One can always see what is in adjacent levels by moving up and down the model hierarchy. Users of a model can view the model at different levels, providing for different levels of abstraction.

2.2.1 Modeling "Top-Down"

To specify models starting with the top level CC often seems to be the most straightforward approach: the model is broken up into a few very general CCs, which in turn can be specified with increasing detail. The number of levels in the HIG and the extend of detail of the CCs at any given level can be chosen freely be the modeler. It is often useful, however, to partition the HIG in ways that correspond to the real world system. A model of a factory, e.g., might have a factory CC as the top level component, on the next level CCs specifying different departments, which could be partitioned into CCs for productions lines, all the way to ACs for conveyors, etc.

MCSs are the hierarchical building blocks of HCFGs. A modeler builds an HCFG by specifying the top level MCS of an AC and its sub-MCSs (if any). HCFGs typically do not have deep hierarchies. Having the ability to specify AC behavior hierarchically is still useful, however, because sub-MCS can be used to (i) partition complex ACs for better manageability and understanding, (ii) separate general from specific behavior, providing a greater potential for reuse behavior, and (iii) hide partial behavior that is trivial or not essential for the understanding of the AC.

Top-down modeling can be a powerful technique to develop models of complex systems, implementing a "divide and conquer" strategy by breaking a large problem up into a number of smaller, more manageable problems. This can be especially useful when the details of lower level Components are not yet clear.

2.2.2 Modeling "Bottom-Up"

Another approach to building the HIG of a model is to start with the ACs used in the model. After specifying ACs, a modeler could encapsulate several ACs into CCs once the model has grown beyond a size that can be easily accom-modated without encapsulation or to demonstrate special relationships among some ACs.

Specifying a HIG bottom-up can be useful when (i) the model is not overly complex, (ii) the model is composed of well known pre-built ACs, (iii) a part of the model is to be studied before being integrated into a larger model, or (iv) the purpose of the simulation is to study the behavior and interactions of basic Components (as opposed the general behavior of a given real-world system).

Specifying models bottom-up has traditionally been the default strategy by simulation systems that rely primarily on pre-built basic elements that can be combined using limited "macro" capabilities.

2.2.3 Mixed Approach

Limited research has shown that for modeling non-trivial real-world problems, a mixed or heterogeneous approach can often be the best way. A modeler could start specifying the HIG from the top down to break the problem up into smaller parts, specify the details of one subcomponent, and if the design works, reuse the specified low-level Components to specify, where appropriate, the remainder of the HIG. The mixed approach was used to specify a non-trivial model of a traffic intersection that is described in Daum (1997).

2.3 Moving Between the Layers

It is important to tightly integrate the modeling layers to avoid the problems of modeling layers that are "too far apart" (Henriksen 1996.) In the HCFG model paradigm, CCs and ACs have the same interface specifications, i.e., the external views have identical properties. (In terms of the object-oriented implementation of ACs and CCs, both derive from a common base class.) This makes it possible to substitute ACs for CCs and vice versa, using the dynamic type feature of experimental frames (Zeigler 1984.)

A modeler could, e.g., have specified several conveyor Components using an AC Type, before realizing that the complexity of the conveyor would have been specified more effectively in a CC. By creating a CC conveyor Type with the same interface, i.e. the same Ports and parameters, as the conveyor AC and changing the type of the conveyor Instances to the new CC Type, a modeler could change the specification of conveyor without having to change any of the (parent) CCs where conveyor Instances are used. Likewise, a modeler could specify conveyor Instances with the Type to be specified in the experimental frame (EF). In the EF, any Type, AC or CC, from a local model library or from across a network, could be specified, as long as the interface is compatible.

3 SCALING

In this section, the scaling of MEs is examined. As modelers build more complex models, scaling becomes an important issue. Scaling is the combination of similar MEs into an ME array. ME arrays provide a powerful mechanism to implement parallelism. For example, a model might contain several parallel servers that share similar characteristics. Specifying these servers as an array of servers simplifies the structure of the model, since common properties of the server elements can be managed in one place, and increases flexibility, since the array size can be changed.

Scaling can be used to create arrays of low level as well as high level MEs. Scaling also includes the connections between MEs, since arrays must be connected differently from single MEs. Because ME arrays are composed of several ME Instances of the same Type, scaling implies ME reuse.

High level MEs that can be scaled in the HCFG model paradigm are Components and MCSs. Channels and Edges can also be scaled to provide ME array interconnectivity. Scaling can be static or dynamic. Static arrays have a fixed size, i.e., the array is specified when the array is created and does not change. The size of dynamic arrays can be specified at runtime and can therefore be changed between simulation runs without recompiling the model.

3.1 Static Arrays

The HCFG model paradigm allows for the specification of ME arrays, MultiChannels, and MultiEdges, providing additional functionality in building non-trivial models. There are two types of arrays: arrays of Components or MCSs and arrays of Channels or Edges, which are also called MultiChannels and MultiEdges, respectively. The default form of arrays is static, i.e., the number of array elements is fixed when the array is created and the number does not change between simulation runs.

Arrays of Components or MCSs consist of a number of Component or MCS Instances that have the same Type and are part of the same parent ME specification.

Component arrays are created similarly to single Components, except that in addition, the modeler must specify the size of the array. To create a static array of Components a modeler would specify a positive integer number.

Component arrays that have the same size can be connected by Channels in the same way single Components are connected. Figure 1 shows two Component arrays, both of size three, which are connected by one bundle of Channels. The visual representation only shows one representative Channel between one representative pair of connected Ports, but the specification will contain three individual Channels, one for each pair of connected Ports, connecting Ports with the same Component index. Table 1 shows how the Ports are connected.

Table 1: Direct Channel Connections Between Component Arrays

Output Ports	connected to	Input Ports
src[0].out	=>	*serv[0].in*
src[1].out	=>	*serv[1].in*
src[2].out	=>	*serv[2].in*

Static arrays of MCSs are specified in a way similar to arrays of Components. Arrays of MCSs that have the same size can be connected similarly as well, except that MCSs have Pins instead of Ports and that MCSs are connected by Edges instead of Channels.

3.1.1 MultiChannels and MultiEdges

In addition to connecting arrays of Components or MCSs with bundles of individual Channels or Edges, the HCFG model paradigm provides the possibility to connect single Components with MultiChannels, i.e., arrays of Channels, and single MCSs with MultiEdges, i.e., arrays of Edges. The distinction between individual Channels that connect arrays of Components and MultiChannels that connect single Components is an aspect of the layered approach to

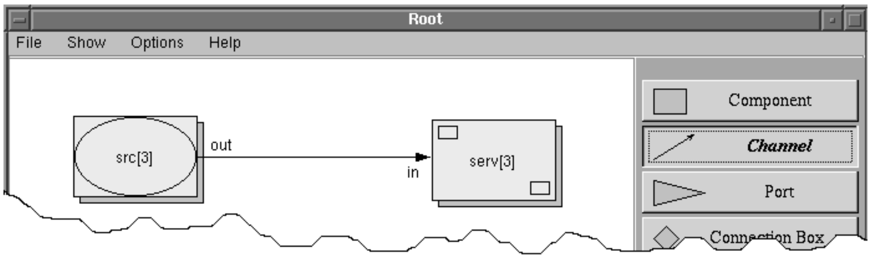


Figure 1: Channel Connection Between Same Size Component Arrays

modeling (see Section 2), where the former property represent parallelism on the Component level, whereas the latter represent parallelism of message traffic.

A modeler can also specify MultiChannels between Component arrays, effectively resulting in two-dimen-sional Channel arrays. The properties of bundled Channels and MultiChannels apply. MultiEdges are specified similarly to (single) Edges, with the added property of array size, which is specified in the same way the size of MultiChannel is specified.

3.1.2 Connection Boxes

One can easily see that the method mentioned above does not work when Component or MCS arrays of different sizes are to be connected or when arrays are to be connected to single MEs. After investigating several options, it was determined that the complex possibilities of $1 - to - n$, $n-to-1$, and $m-to-n$ connections could not be represented graphically in a way that was simple, concise, and easy to model and understand. This becomes especially clear when one considers very large arrays and the possibilities for connecting them, which are even more numerous.

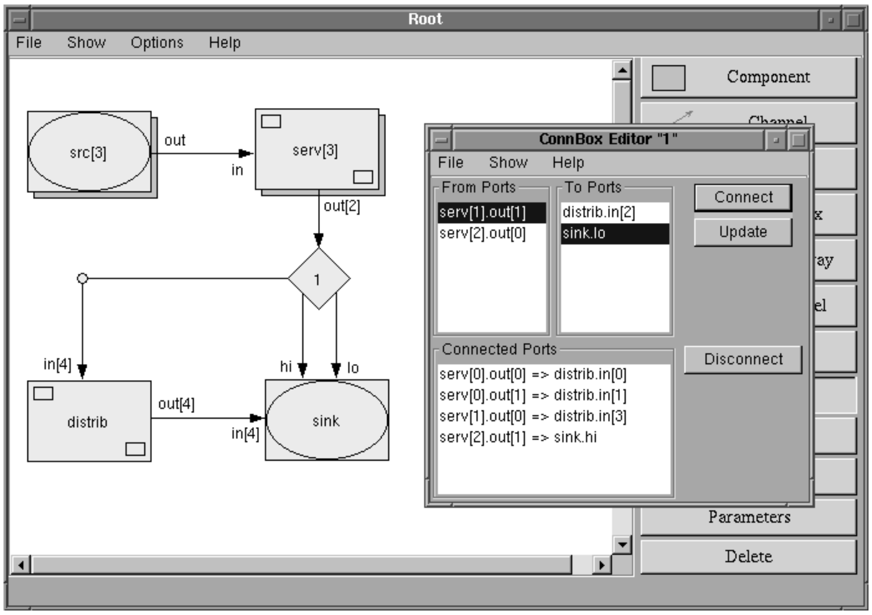
To solve these problems, modelers can use *Connection Boxes*. Connection Boxes are helper elements, i.e., they are not part of a Type specification, but rather serve as a utility to simplify the modeling process and also make a specification easier to read. A modeler can specify Connection Boxes, which are diamond shaped and each have an auto generated number, using the *Connection Box* tool, which is available in both the CC and the MCS specification window. A modeler can connect any number of Channels and/or MultiChannels to a Connection Box, regardless of array sizes. Clicking on a Connection Box displays a new window that initially shows a list with the unconnected Output Ports, another list with the unconnected Input Ports and a third list for connected pairs of Ports, which is initially empty. Connecting and disconnecting Ports can be achieved by simple point-and-click operations. Figure 2 shows an example of a CC Type specification, which uses a Connection Box to connect various Channels and MultiChannels, as well as the opened Connection Box window, with all but two pairs of Ports already connected.

3.2 Dynamic Arrays

Limited research has shown that having the ability to change the size of arrays in the EF would (i) significantly increase the potential for ME reuse and (ii) simplify the changing of model conditions between simulation experiments. A model might contain, e.g., an array of parallel servers. Increasing or decreasing the number of servers by changing the size of the array in the EF could yield important data desired in the experiment, all without having to recompile the model. The HCFG model paradigm provides for the possibility to change the sizes of arrays through dynamic arrays. See Daum 1998 for a an implementation of EFs for HCFG models.

3.2.1 MEs

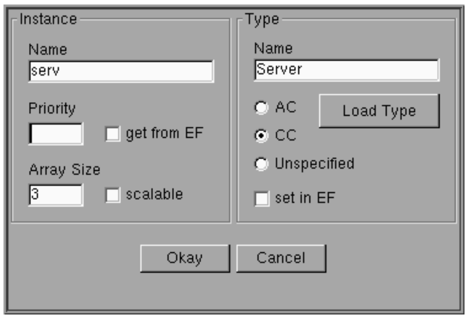
Dynamic arrays of Components or MCSs are specified similarly to static arrays of these MEs (see Figure 3), except



From Ports	To Ports
serv[1].out[1]	distrib.in[2]
serv[2].out[0]	sink.lo

```
serv[0].out[0] => distrib.in[0]
serv[0].out[1] => distrib.in[1]
serv[1].out[0] => distrib.in[3]
serv[2].out[1] => sink.hi
```

Figure 2: Channel Connections through a Connection Box



Instance — Name: serv; Priority: (blank), get from EF: unchecked; Array Size: 3, scalable: unchecked
Type — Name: Server; ○ AC, ● CC, ○ Unspecified; set in EF: unchecked

Figure 3: Component Array Dialog Box

that a modeler would mark the *scalable* checkbox in the dialog box. Modelers have three options for specifying a value in the *Array Size* field, they can:

1. Enter the name of an integer variable from the parent Type. This would cause the creation of an array with the size of the value of this variable at runtime. In the ME Graphical User Interface (GUI) window, the Instance name prefix will be shown with the name of the variable, enclosed in square brackets, e.g., *serv[k]* to denote an array of *serv* Instances with the size of *k* at runtime. The value could be set in the EF, or initialized with a parameter, which can be especially useful when several Instances of the parent Type exist and Instance specific parameters are passed down the Component hierarchy.
2. Leave the size filed blank. This will create an integer variable with an identifier that combines the name of the instance with the suffix `_size` and add an entry to the *Variable* section of the EF in the form

   parentTypeName`.`*instanceName*`_size`,

   the value of which will be used to determine the size of the array at runtime. In the ME GUI window, the Instance name prefix will be shown with square brackets, e.g., *serv[]*, to indicate that the size of the array has not yet been specified.
3. Enter a positive integer number. This will create a variable and EF entry as in 2., but also set a default array size. In the ME GUI window, the Instance name prefix will be shown with the default size of the array and a ˜ symbol, enclosed in square brackets, e.g., *serv[3˜]*, to indicate that the default size can be changed in the EF. At runtime, the array will be created with a size specified in the EF or, if no value is found, with the default value that was specified.

Since the size of a dynamic array is not known before it is specified in the EF, values for all parameters of the array elements have to be specified in the EF as well. The EF specification GUI contains a section *Dynamic Parameters*. When this section is selected, the *Variable* section of the EF is searched and if a value for an array size is found the corresponding number of parameter entries is generated. If no value has been found, a warning is message is displayed.

3.2.2 MultiChannels and MultiEdges

MultiChannels and MultiEdges can be specified as dynamic arrays by marking the *scalable* checkbox in the dialog box used to specify them. The options for the value entered in the *Array Size* field, if any, are essentially the same as for dynamic ME arrays. An auto-generated variable would have the form

parentTypeName`.`*instanceName*
`.`*outputPortOrPinName*`_size`.

Note that changing the size of a MultiChannel changes the size of its Port arrays. Since Ports are part of the Component interface, i.e., they are visible both from the outside and the inside of the Component, dynamic MultiChannels effectively make the Type specifications of all connected Components dynamic as well. This capability provides a powerful mechanism for model customization and reuse.

Dynamic bundles of Channels and Edges, MultiChannels, and MultiEdges can only be connected to dynamic Connection Boxes, which look like static Connection Boxes, except that their connections can only be specified in the EF. The EF contains a section *Dynamic Connections*, which contains the entries for all of the dynamic Connection Boxes. When a Connection Box is selected, the software will search the *Variable* section for the needed values and, if they are found, display the correct number of Ports or Pins to be connected, including the static Ports and Pins that are connected to the Connection Box, if any.

4 MODEL ELEMENT REUSE

In this section, an argument for ME reuse is made. Different kinds of reuse are introduced and the type–instance relationship as a form of reuse is discussed.

4.1 Kinds of Reuse

One of the most important properties of a modern modeling and simulation system is the ability to reuse parts of the model specification. Reuse can be realized at different levels.

Low level reuse is the reuse of the most basic model building blocks of a simulation system such the statements of a text-based simulation language or the low level elements of a GUI-based simulation system. *High level* reuse is achieved by systems that provide some mechanism for combining several basic model building elements that are often used in the same context into higher level elements that can be reused.

When investigating reuse, one has to consider the origin of reusable model building elements. Reuse of *predefined elements* is provided by most simulation systems. Many systems use a set of predefined building blocks as the core of the modeling capability. Reuse of *custom built elements* is provided only by some simulation systems, as not all systems support the specification of building blocks by the modeler.

Another important distinction has to be made as to the scope of the reuse. Reuse as *individual elements* is the reuse of building blocks independent from each other, i.e., identifiers, parameters, and interactions with the rest of the model have to be specified separately. Reuse as *array elements* is the reuse of building blocks as array elements of the same type. Identifiers, parameters, and interactions with the rest of the model can be specified incorporating array indices. Reuse *within the same model* provides for the reuse of frequently used atomic building blocks as well as larger parts of the model (if high level components are supported) in the same model. It may be necessary to assign unique identifiers to distinguish several instances of the same building block. Reuse *across different models* requires a mechanism such as a component library for preserving custom built element specifications independently from model specifications.

4.2 Instances and Types

The HCFG Model paradigm distinguishes between “Types” of MEs, which represent the specifications of MEs, and “Instances” of MEs which represent concrete representations of ME Types in a model. This model strongly corresponds to the object-oriented paradigm (Eckel 1998) of objects and classes, with Types essentially being classes and Instances being objects.

A Type specifies the contents or behavior of a ME. CC types and MCS types, e.g., can contain child Components and child MCSs, respectively, thus providing for hierarchy, among other ME attributes. A model would typically contain only one ME Type for any desired part of behavior that could be reused in different instances throughout the model. A Type specifies what can be called the *internal view* of a ME, which can be accessed only through the interface of the ME. An interface contains the parameter list of the ME and can also contain other ME attributes, e.g., Ports.

An Instance is a specific representation of ME Type, specifying the location of the ME in the model, a name so that the instances can be uniquely identified in the model, and provides access to the interface of the ME for the interaction of the ME with the outside world. An Instance specifies what can be called the *external view* of a ME. Following the principle of object encapsulation, other MEs can access the Type of a ME Instance only through the ME interface. Creating several Instances of a Type is the basic form of reuse in the HCFG Model paradigm.

MEs that can be reused by creating multiple Instances from Types include Components, i.e., CCs and ACs, MCSs, Conditions, i.e., Time Delay Conditions and Boolean Conditions, and Events. ME Arrays can be similarly reused.

Creating ME Instances from scratch and optionally reusing the Types that are automatically generated in the process is a feasible approach when specifying noncomplex models or when modeling *top-down*, i.e., starting with high-level components that can be subsequently broken down into subcomponents to specify more detailed behavior.

Another way of modeling is *bottom-up*, where a modeler first builds specific components that specify a certain behavior that is known to be needed in the model and then incrementally combines these components into higher level components. In such a case, it is possible to specify ME Types that do not yet have any Instances in the model. Once a modeler has specified the basic Types needed in the model, they can start creating Instances of these Types, combine Instances into higher level components, thus building the model tree.

5 SUMMARY

A brief overview of the HCFG model paradigm was presented. The paradigm tightly integrates hierarchical modeling, ME scaling, and ME reuse. Hierarchical modeling greatly increases the manageability of large models. Complex problems can be broken down into smaller problems which can be addressed separately. Different levels of abstraction can provide for a more natural way of modeling and help to focus on different degrees of detail when using a model.

Scaling of MEs provides for combining single MEs into ME arrays. ME arrays provide a straightforward and flexible way to model parallelism. The connection of arrays of different sizes has proven to be not trivial. Connection boxes are used to connect different size arrays in an intuitive way. Dynamic arrays provide for easy scalability of model parts between simulation runs.

Reuse of MEs allows for the repeated use of ME specifications by creating several instances from a single ME type. Both MEs built from scratch by the modeler and pre-built MEs from a library can be reused in the same, straightforward way. ME reuse can significantly reduce model specification time and effort.

The HCFG model paradigm and HiMASS-j exploit the potential of hierarchical modeling, scaling, and reuse, which are tightly integrated into the system. We found that the inter-operability between these techniques is particularly interesting. For example, hierarchical modeling allows for the reuse not only of the most basic MEs, but also very high level Components of arbitrary complexity. The scaling of a hierarchical component scales all its subcomponents, which in turn can be arrays, as well. Scaled MEs can be reused with a different size, providing powerful capabilities for modeling parallelism.

REFERENCES

Cota, B. and R. Sargent. 1992. A modification of the process interaction world view. *ACM Trans. Model. Comput. Simul.*, 2, 2, 109–129.

Daum, T. 1998. *An Investigation into Specifying HCFG Models using Visual Interactive Modeling*. Graduate Thesis. Otto von Guericke University in Magdeburg.

Daum, T. and R. Sargent. 1997. A Java Based System for Specifying Hierarchical Control Flow Graph Models. In: S. Andratdottir, K.J. Healy, D.H. Withers, and B.L. Nelson, eds., *Proc. of the 1997 Winter Simulation Conference*, 150–157.

Daum, T. 1997. An HCFG Model of a traffic intersection specified using HiMASS-j. In: S. Andratdottir, K. Healy, D. Withers, and B. Nelson, eds., *Proc. of the 1997 Winter Simulation Conference*, 158–165.

Eckel, B. 1998. *Thinking in Java*. Upper Saddle River, New Jersey: Prentice-Hall.

Fritz, D. and R. Sargent. 1995. An overview of hierarchical control flow graph models. In: C. Alexopoulos, K. Kang, W. Lilegdon, and D. Goldsman, eds., *Proc. of the 1995 Winter Simulation Conference*, 1347–1355.

Henriksen, J. 1996. An Introduction to SLX. In: J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, eds., *Proc. of the 1996 Winter Simulation Conference*, 468–475.

Sargent, R. 1997. Modeling queueing systems using hierarchical control flow graph models. *Mathematics and Computers in Simulation*, 44, 3, 233–249.

Zeigler, B. 1984. *Multifacetted Modelling and Discrete Event Simulation*. London: Academic Press.

AUTHOR BIOGRAPHIES

THORSTEN DAUM is a senior software engineer with ObjectGuild Inc. in San Jose, California. He holds a graduate degree in computer science with a focus on simulation from Otto von Guericke University in Magdeburg. His interests include the development of visual interactive modeling systems for simulation and Java software. He has been a visiting researcher with the Simulation Research Group and CASE Center at Syracuse University.

ROBERT G. SARGENT is a Research Professor/Professor Emeritus in the L. C. Smith College of Engineering and Computer Science at Syracuse University. He received his education at the University of Michigan. Dr. Sargent has served his profession in numerous ways and has been awarded the TIMS (now INFORMS) College on Simulation Distinguished Service Award for long-standing exceptional service to the simulation community. His research interests include the methodology areas of modeling and discrete event simulation, model validation, and performance evaluation. Professor Sargent is listed in *Who's Who in America*.