Syracuse University
SURFACE

Electrical Engineering and Computer Science | College of Engineering and Computer Science

1995

Verification And Validation Of Simulation Models

Douglas G. Fritz
Syracuse University, Simulation Research Group

Robert G. Sargent
Syracuse University, Simulation Research Group

Thorsten Daum
University of Magdeburg, Department of Computer Simulation and Graphics

Follow this and additional works at: https://surface.syr.edu/eecs

 Part of the Computer Sciences Commons

Recommended Citation
Fritz, Douglas G.; Sargent, Robert G.; and Daum, Thorsten, "Verification And Validation Of Simulation Models" (1995). *Electrical Engineering and Computer Science*. 113.
https://surface.syr.edu/eecs/113

This Article is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

AN OVERVIEW OF HI-MASS
(HIERARCHICAL MODELING AND SIMULATION SYSTEM)

Douglas G. Fritz
Robert G. Sargent

Simulation Research Group
Syracuse University
439 Link Hall
Syracuse, New York 13244, U.S.A.

Thorsten Daum

Department of Computer Simulation
and Graphics
University of Magdeburg
D-39016 Magdeburg, GERMANY

ABSTRACT

The Hierarchical Modeling and Simulation System (HI-MASS) is a prototype modeling and simulation system that supports modeling based on the Hierarchical Control Flow Graph Model paradigm and simulation execution using a sequential synchronous simulation algorithm. The prototype is an object oriented C++ based system designed for a Unix environment and implemented using freely available software tools. Models are specified using two complementary hierarchical model specification structures, one to specify the components which comprise a model and how those components are interconnected, and the other to specify the behaviors of the individual components. A graphical user interface provides for component and interconnection specification using visual interactive modeling. Behavior specifications are constructed using C++ classes and functions provided by HI-MASS.

1 INTRODUCTION

This is a companion paper to "An Overview of Hierarchical Control Flow Graph Models" (Fritz and Sargent 1995) contained in these proceedings. It is assumed that a reader of this paper is familiar with that paper.

The Hierarchical Modeling and Simulation System (HI-MASS) is a prototype simulation system that supports the modeling and execution of discrete event simulation models based on the Hierarchical Control Flow Graph Model paradigm and uses a sequential synchronous simulation execution algorithm.

Hierarchical Control Flow Graph Models use two complementary types of hierarchical model specification structures. The first type of specification structure, called a Hierarchical Interconnection Graph (HIG), is used to specify the atomic and coupled components that comprise the model and how those components are interconnected. HI-MASS provides a Graphical User Interface (GUI) for specification of the HIG using "visual interactive modeling". The second type of specification structure, called a Hierarchical Control Flow Graph (HCFG), is used to specify the behaviors of the individual Atomic Components (AC's) of the model. HCFG's are currently specified using a text editor to enter C++ programming language source code based on C++ classes and functions provided by HI-MASS. (GUI support for HCFG specification is planned for a future version of HI-MASS.)

An HCFG Model specification consists of one HIG plus an HCFG behavior specification for each type of AC used in the model. The HIG specification is transformed into C++ code using utility programs provided by HI-MASS. The elements of the model specification are then compiled and linked with the HI-MASS object library to form an executable model. The executable model is then combined with experimental conditions specified by an "experimental frame" to produce a simulation run. A high level overview of this process is given in Figure 1.

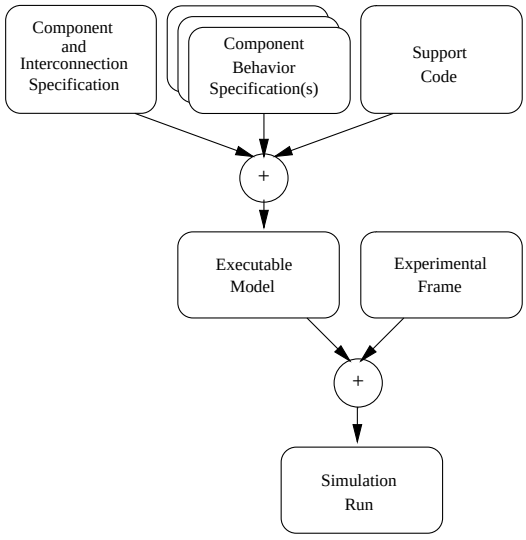

Figure 1: High Level Overview

HI-MASS is an object oriented C++ based system which was developed specifically for a Sun SPARC workstation running the SunOS (Unix) operating system. HI-MASS makes extensive use of object oriented programming features supported by C++, such as encapsulation, inheritance, polymorphism, genericity, and overloading (Stroustrup 1991). HI-MASS was implemented using the freely available GNU C++ compiler (g++) and C++ library (libg++). The GUI used for visual interactive modeling was implemented using the freely available InterViews C++ user interface toolkit. HI-MASS should compile and run on any system on which this compiler, library, and toolkit are installed. In addition to the Sun SPARC, we have run HI-MASS on an IBM RS/6000, a DEC Alpha, and on Intel 486 and Pentium based personal computers.

A user of HI-MASS needs to be familiar with modeling using the HCFG Model paradigm and software development using the C++ programming language in a Unix based environment. An 82 page User's Guide (Fritz, Daum, and Sargent 1995) which includes several examples has been developed for HI-MASS

The remainder of this paper is organized as follows. Sections 2 and 3 describe, respectively, the specification of the HIG and the HCFG behaviors. Section 4 presents an overview of the use of experimental frames. Sections 5 and 6 describe the construction and the execution of a model, respectively. Section 7 summarizes this paper.

2 COMPONENTS AND CHANNELS

A HIG is used to specify the components which comprise a model and how those components are interconnected (i.e., the channels). A HIG is a hierarchical structure of components which is defined by a set of Coupled Component Specifications (CCS's), each of which specifies the internal view for a coupled component type. (The internal view of an AC is a behavior specification and is not part of the HIG.)

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. If multiple model components are "instances" of the same type of component, then those components all share the same type definition. A CCS for a coupled component specifies a set of subcomponents and a set of channels which define the routing pattern for all intercomponent messages between the subcomponents and between the subcomponents and the "outside world" (i.e., the external ports of the enclosing component). The coupled component that encloses the entire model (i.e., the root node of the associated HIG tree) is the only component in a model that has no external ports.

The HIG for a model is constructed by specifying the set of CCS's for the coupled component types that are contained in the model using the HI-MASS GUI. The GUI uses two different file formats for CCS's: visual and structural. The visual format is used by the GUI to define the visual appearance of the modeling elements of the CCS, such as shapes, locations, and relations. The structural format defines the CCS elements such as subcomponent name/type pairs, port identifiers, and port interconnections and is used for the generation of the C++ code for model construction. The GUI requires only the visual format for saving and loading CCS's. When the HIG is complete, the GUI is then used to generate the structural format for each CCS in the HIG.

2.1 Visual Interactive Modeling

HI-MASS supplies a GUI which allows a modeler to construct the HIG using visual interactive modeling. A HIG is constructed using the GUI's "point and click" environment to specify the set of CCS's in the HIG. A screen dump of a GUI window is shown in Figure 2.

In Figure 2 the top line (containing "HI-MASS HIG Graphical Interface") is the X11 title bar. The next line shows that the component type name for this CCS is "System". All components of type "System" in the model will share this definition. Clicking on the type name will pop up a dialog box which can be used to modify the component type name. Just below the type name is a menu bar with three "pull down" menus: "File", "Generate", and "Info". The "File" menu contains operations that allow a modeler to save and restore CCS's. The "Generate" menu allows a modeler to generate the structural format files when the HIG is completed. The "Info" menu displays the GUI development team and copyright notice.

To the right of the "canvas" area is a set of "tool" buttons. A user "selects" a tool by clicking on the appropriate button, and then "applies" the tool in the canvas area. For example, the "Component" tool allows the modeler to create a new component. After selecting the "Component" tool, the modeler "clicks" on the canvas at the desired location for the new component. A "dialog box" then pops up for the user to enter the name and type for the new component. The GUI will then verify that the name is valid (e.g., no other component has the same name), and if so, it will then place the new component (represented by a rectangle) on the canvas. Both the instance name and the type name of a component can be modified. The names shown on components in the GUI's canvas area are the instance names. The "Component Array" tool creates a homogeneous array of components. (A

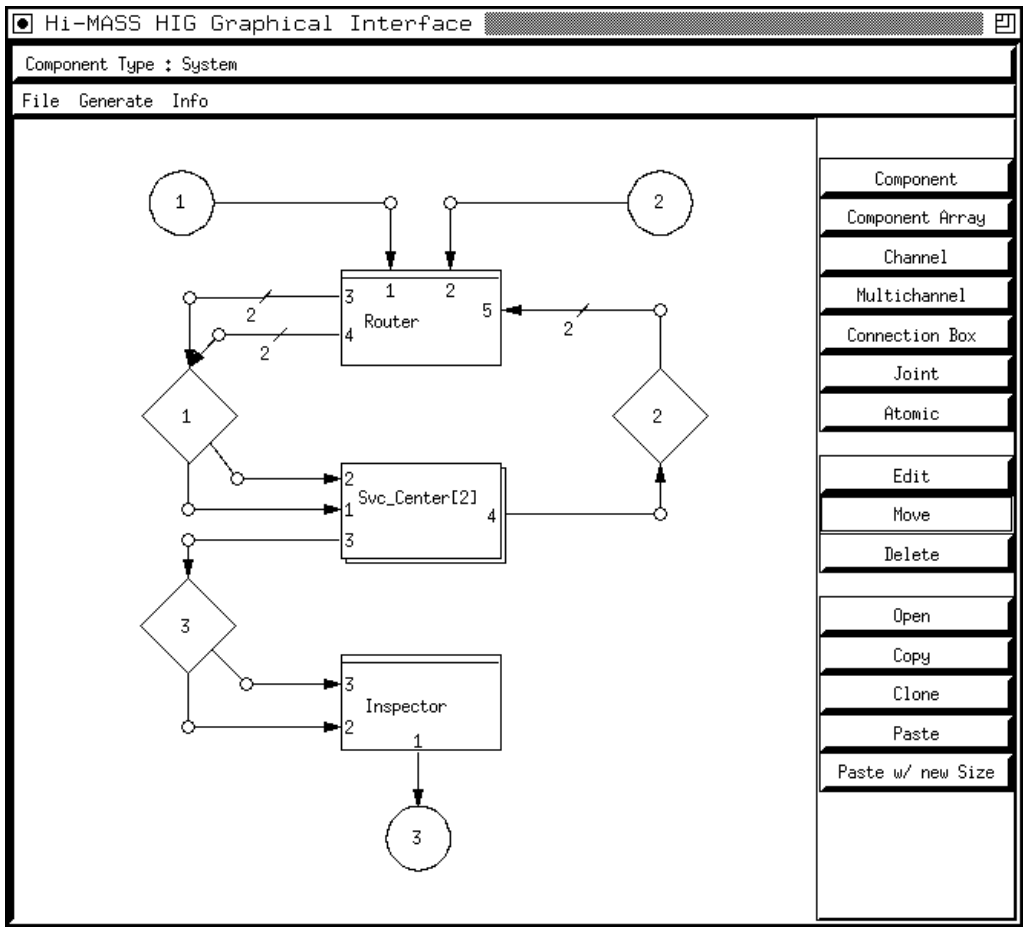

Figure 2: Graphical User Interface Window

component is equivalent to a component array of size one.) Figure 2 shows two components, "Router" and "Inspector", and one component array of size two, "Svc_Center[2]", for a total of four components. Individual elements of a component array are referenced using a zero based index (e.g., "Svc_Center[0]" and "Svc_Center[1]").

An atomic component is visually distinguished by an additional horizontal line near the top of the component (added using the "Atomic" tool). In Figure 2 "Router" and "Inspector" are AC's, and the two elements of the component array are coupled components.

The "Channel" tool is used to create and/or connect component ports. A channel is represented graphically as a polyline (open polygon) with its direction indicated via an arrow. "Joints" or "pivot points" (small circles) in a channel's graphical representation are used to allow the modeler to position the channels. The "Joint" tool allows a modeler to add additional "joints" to existing channels. When channels are created, ports are automatically added to the appropriate components and unique port identifiers (numbers) are generated for the newly created ports. The "Multichannel" tool is used to create a "bundle" (array) of channels of a size specified by the modeler; the bundle size is indicated on the multichannel. Bundles of channels create port arrays of the specified (bundle) size which are referenced using a zero based index.

Not all connections can be clearly represented using a purely graphical notation (e.g., connections between component arrays of different sizes). The "Connection Box" tool is used to specify such connections. A connection box (represented by a diamond) is a place-holder which is linked to a textual interconnection representation. By selecting the "Edit" tool and clicking on a connection box, the GUI invokes an external text editor for the modeler to specify the interconnections. The GUI will automatically generate port identifiers for all channels that are connected to the connection box and place them in the connection file for the modeler to edit, thus a modeler has only to "cut" and "paste" to specify the connections. The

connection syntax is straightforward.

The "Edit", "Move" and "Delete" tools provide common editing capabilities. The "Edit" tool (in addition to its use with connection boxes) is used to modify component names.

The "Open" tool allows the modeler to open a coupled component to view and/or specify the coupled component's internal view (i.e., its CCS). When the modeler clicks on a coupled component, another GUI window containing the internal view of the selected component automatically opens. For example, if the CCS in a GUI window is as shown in Figure 3 and we "Opened" the "System" component, we would then get a second GUI window as shown in Figure 2. Notice that the port identifiers, "1", "2", and "3" from the external view of the "System" component become the external port identifiers in the internal view (shown in the circles in Figure 2).

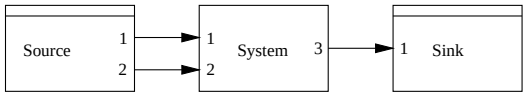

Figure 3: Top Level CCS

The "Copy", "Clone", "Paste", and "Paste w/new Size" tools are used to create copies and clones of existing components and/or component arrays. A copy creates a new instance of an existing component type whereas a clone creates a new component type.

2.2 Generate and Flatten

Once the HIG specification is complete, the GUI's "generate" function is then used to generate the set of component specifications (structural representations) required by later stages of the HI-MASS model construction process. The structural representation consists of one component specification for each type of component used in the model.

In HI-MASS, the HIG is then "flattened" into an Interconnection Graph (IG) using the HI-MASS "flatten" utility program. This flattening removes the coupled components from the model while preserving the interconnection information of the original HIG. The output of the flattener is a single CCS (the IG) in which the subcomponents consist of all the AC's of the original HIG. In an HCFG Model all message traffic originates and terminates at AC's, but the messages may pass through one or more intermediary coupled component ports between the two AC ports. In the IG all the coupled components have been removed and the interconnections are directly between the original AC's of the HIG.

3 BEHAVIOR SPECIFICATION

The behavior for each type of AC is specified using an HCFG. An HCFG is a rooted tree structure in which the nodes represent Macro Control States (MCS's). The MCS is the basic building block for behavior specification. In the current version of HI-MASS a modeler specifies the behavior of each MCS via object oriented C++ programming language code based on classes and functions provided by HI-MASS. An AC behavior specification (i.e., the HCFG) is defined by the set of MCS specifications that constitute the nodes of the AC's HCFG tree.

3.1 Macro Control States

HI-MASS provides a C++ base class "MCS" from which all MCS classes in a model are derived. This base class defines the attributes and behavior common to all MCS's. A modeler must create a MCS class for each type of MCS used in a model. Each type of MCS specifies any required parameters in its C++ constructor declaration. (A constructor for a C++ class is a special function that specifies how to construct an object of that class.) Parameters that an MCS might require include initial values for local variables, pointers to external functions and variables, and pointers to AC ports. All MCS's of the same type throughout a model share the same class definition. Attributes common to all MCS's include a "name", a "type name", a set of pins, a set of control states, a set of edges, and a set of child MCS's.

The MCS class provides "helper" class member functions which a modeler can use in constructing a MCS. These helper functions allow a modeler to easily create objects such as pins, control states, and edges. Child MCS's contained within a MCS must be created by directly invoking the child MCS's constructor as MCS instantiation (construction) may require model and/or context dependent parameters.

We illustrate the use of these helper functions by constructing a MCS of type "(Simple)". This MCS, shown in Figure 4, can be specified using C++ code resembling that shown in Figure 5. The helper functions were designed so that even those who are unfamiliar with C++ syntax should be able to understand the basic operation performed by each line of code in Figure 5.

Lines 1 through 4 of Figure 5 create and add the (external) pins and control states to the MCS. Line 5 creates the child MCS "child1" and line 6 adds it to the current MCS (so that its pins can be referenced when adding edges). Lines 7 through 12 add the edges which interconnect pins and control states of the MCS. The first two parameters specify the origin

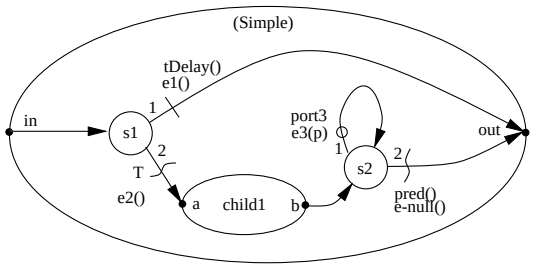

Figure 4: A Simple MCS

```
 1) Pin*       in     = add_Pin("in");
 2) Pin*       out    = add_Pin("out");
 3) CtrlState* s1     = add_CtrlState("s1");
 4) CtrlState* s2     = add_CtrlState("s2");
 5) MCS* child1 = new MCS_Child("child1",...);
 6) add_MCS(child1);
 7) add_Edge(in,s1);
 8) add_TimeEdge(s1,out,tDelay,e1,1);
 9) add_BoolEdge(s1,child1->get_Pin("a"),0,e2,2);
10) add_Edge(child1->get_Pin("b"),s2);
11) add_PortEdge(s2,s2,&port3,e3,1);
12) add_BoolEdge(s2,out,pred,0,2);
```

Figure 5: Code for a Simple MCS

and termination points for the edge. Edges originating from pins have no other attributes. For those edges originating from control states, the third parameter specifies the condition, the fourth parameter specifies the event, and the fifth parameter specifies the edge priority. The condition and event parameters are pointers to functions that are called whenever the edge's condition attribute is tested or its event is executed (during an edge traversal). This method of passing a pointer to a function that is to be called back at a later time is referred to as a "call back".

As an example of the helper function semantics, line 9 in Figure 5 adds a TrueEdge from control state "s1" to the input pin "a" of the child MCS "child1". The "a" pin belongs to the child MCS "child1" and thus we ask "child1" for a handle (pointer) to its input pin named "a" using the construct "`child1->get_Pin("a")`". The third parameter of "`add_BoolEdge()`" is either a pointer to a predicate function which returns **true** or **false**, or a value "0" for a TrueEdge. (We are able to use a pointer value of zero "0" for a TrueEdge because "0" is an invalid pointer value in C++.) The fourth parameter is a pointer to the event function that is executed when the edge is traversed. The fifth and last parameter is the edge priority. The TrueEdge specified on line 9 has a priority value "2".

We say that the code in Figure 5 "resembles" that required in HI-MASS because the code shown assumes that the condition and event functions are regular C++ functions. However, in HI-MASS, the condition and event functions are encapsulated within the MCS in which they are defined as class "member functions". Member functions require a different method for use as "call back" functions. In HI-MASS we use a C++ idiom, known as "Functors", that allows "type safe" call backs of member functions using objects that act like functions. (See Hickey (1995) for a discussion on "Functors".)

In addition to the specification of the MCS graph (pins, control states, MCS's, and edges) given in Figure 5, we must also specify each condition and event function used in the MCS and any member variables that those functions may require. Time delay functions (used by TimeEdges) are functions that return a non-negative time delay value, and boolean predicate functions (used by BoolEdges) return **true** or **false**. PortEdges require a pointer to an associated input port (instead of a function) which is interrogated for its empty/non-empty status. Event functions for TimeEdges and BoolEdges are functions that take no parameters. These functions are called whenever the simulation execution algorithm traverses the associated edge. The event function for a PortEdge is a function that takes as its only parameter, a pointer to the associated port. For example, the event associated with the PortEdge referenced in line 11 of Figure 5 might be defined as in Figure 6.

```
void e3(InPort* p) {
  Msg* m = p->receive(); // get msg from port
  message_count++;       // increment count
  delete m;              // destroy message
}
```

Figure 6: PortEdge Event "e3" Definition

An overview has now been given on the specification of MCS's in HI-MASS. We next look at the messages that are passed between the AC's of the model.

3.2 Messages

All intercomponent messages in HI-MASS are derived from a C++ base class "Msg" supplied by HI-MASS. Message passing is handled in HI-MASS by dynamically creating messages, passing pointers to the messages between AC's, and having the receiving AC destroy the dynamically allocated messages after all required information has been extracted from them.

All messages in HI-MASS have a timestamp which is set to the local simulation time of the sending AC when the message is transmitted. Messages may also have other attributes which are used to carry additional information between AC's. The sender and receiver *must* use compatible message types.

HI-MASS provides a message class "Msg_Proto" (derived from base class "Msg") which contains six attribute fields: two integer, two floating point, and two String. If all messages used in a model are of class "Msg_Proto" then message compatibility between the sending and receiving message types is assured. (HI-MASS allows a modeler to define other message types, but this requires the modeler to assume responsibility for message compatibility.)

4 EXPERIMENTAL FRAME

HI-MASS supports the use of experimental frames. The Experimental Frame (EF) concept separates a model's definition from the set of model parameters used for a specific execution run of the model. This allows a modeler to modify such items as initial conditions, seed values for random number generators, desired data collection, and termination conditions for a specific simulation run without modifying the model itself (which would require an edit, compile, and link cycle). HI-MASS reads experimental frame information from two text files during its initialization phase prior to constructing the model objects (e.g., the AC's and MCS's). One EF file contains information that specifies the initial control state for each AC in the model, and the other EF file contains information that specifies the initial values for those model variables that have explicit support for EF initialization. Setting the initial control state for each AC is handled entirely by HI-MASS support code (contained in HI-MASS base classes) and thus involves no modeler action other than selecting the initial control state for each AC. Support for setting initial values for model variables must be explicitly supplied by the modeler; HI-MASS provides helper functions that simplify this task. A modeler will generally have several sets of EF files associated with a model (one set for each experiment). The EF files used for a specific simulation run are specified via command line options when invoking the simulator.

We have now discussed the elements which comprise a HI-MASS model. In the next section we look at how these elements fit together.

5 BUILDING A MODEL

An HCFG Model requires one HIG plus an HCFG for each type of AC used in the model. In this section we show how those two types of model specifications are combined, along with HI-MASS support routines, to build an executable model. This process is shown in Figure 7. The steps a modeler takes to construct a model are shown in solid boxes and the supporting elements of HI-MASS that are utilized by the modeler are shown in "dashed" boxes. Creating the HIG using the GUI and flattening the HIG into an IG were covered in Section 2, an overview of MCS specification was given in Section 3, and the use of experimental frames was covered in Section 4. We now cover the remaining steps for constructing a model.

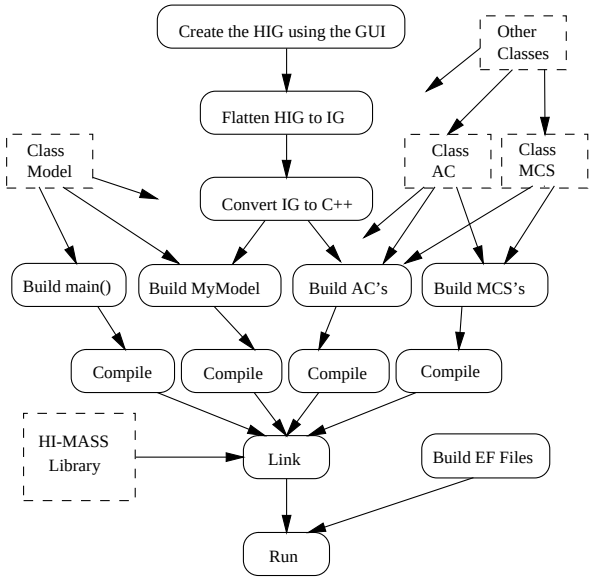

Figure 7: HI-MASS Modeling Tasks and Elements

5.1 Classes and Functions

A HI-MASS model (in its C++ representation) has one object of type "Model", one object of type "AC" for each AC, and one object of type "MCS" for each MCS in the model. The "Model" object is defined by a class (e.g, "MyModel") that is derived from the HI-MASS base class "Model". (Class inheritance defines an "is-a-kind-of" relationship from the derived class to the base class, thus "MyModel" "is-a-kind-of" "Model".) Each type of AC in the model is defined by a class derived from base class "AC", and each type of "MCS" in the model is defined by a class derived from base class "MCS". The relationships between the "Model", "AC", and "MCS" objects and the partitioning of information from the two types of specification structures (HIG and HCFG) between these classes are shown in Figure 8.

The "Model" object constructs each AC in the model and then interconnects the AC ports as specified by the IG. Each "AC" constructs its set of input ports, output ports, and the top level MCS of its HCFG tree. Each MCS constructs its pins, control states, child MCS's, and edges, and defines all condition and event functions required by its edges. (Since MCS's recursively construct any child MCS's contained within them, an HCFG tree is uniquely defined

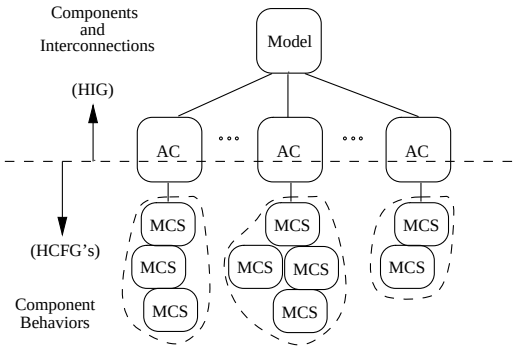

Figure 8: Specifications and Classes

by specifying its top level MCS. Thus, an AC object completely defines its behavior by specifying only the top level MCS of its HCFG tree.)

Every C++ program also has a function "main" which is the main entry point where program execution begins. All the C++ code in a HI-MASS model that is not contained within one of the C++ classes (as member functions) is contained within the "main" function. A typical main routine for a HI-MASS model (as shown in Figure 9) is a very simple routine that performs only three functions: (1) it creates the "Model" object, (2) it tells the "Model" object to initialize the model, and (3) it tells the "Model" object to execute (simulate) the model. All "main" routines for HI-MASS models are the same, except for possibly the type name of the "Model" object (e.g., "MyModel").

```
main(int argc, char *argv[]) {
    Model* model = new MyModel(); // create
    model->init(argc,argv);       // initialize
    model->execute();             // execute
}
```

Figure 9: Function "main()"

5.2 Incorporating the IG

The HIG specifies the components and interconnections for a model. The HIG is then flattened into an IG. Then the following information is extracted from the IG for model specification: (1) a list of all AC's contained in the model and the associated type of each, (2) a list of all AC port to port interconnections (i.e., the channels), and (3) a separate list of the set of input and output ports for each type of AC used in the model. The types of information extracted from the IG are shown in Figure 10. (The "n" on the arrow from the IG to the "List(s) of Ports" in Figure 10 indicates that there are "n" lists of ports, where "n" represents the number of AC types used in the model.)

As stated in the previous subsection, a modeler must create a class (e.g., "MyModel") that is derived from the base class "Model". The only difference between base class "Model" and class "MyModel" is that class "MyModel" provides definitions for two functions; one function creates the AC objects in the model and the other function specifies the AC port interconnections. A HI-MASS utility program "`ig-to-c++`" converts the information from the IG into the C++ form required by these functions in the "MyModel" class. A modeler can simply copy a sample "MyModel" class supplied by HI-MASS and then "cut" and "paste" the AC and interconnection information extracted from the IG into the appropriate "MyModel" functions using a text editor. No other modeler input is required for a class "MyModel".

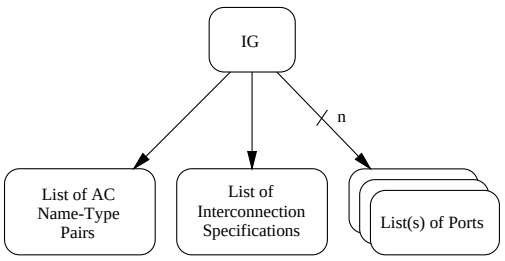

Figure 10: Information from the IG

A modeler also creates a class for each type of AC used in the model. All AC classes are derived from the HI-MASS base class "AC". These classes are identical except for three elements: (1) the set of input and output ports, (2) the behavior specification, and (3) the set of AC (local to the AC) attributes. The set of input and output ports for an AC type is extracted from the IG using the "`ig-to-c++`" utility. The behavior specification is generated by creating the MCS for the root node of the AC's HCFG tree (child nodes (MCS's) in the HCFG tree are recursively generated). The behavior specification in an AC class is typically specified via a single line of code. The AC attributes vary depending upon the type of AC; some attributes (such as a local simulation clock) are standard among all AC's while others must be explicitly specified by the modeler.

5.3 Compiling and Linking

A model consists of a function "main()", a class "MyModel", a class for each type of AC used in the model, and a class for each type of MCS used in the model. The function "main()" and the "Model", "AC", and "MCS" class definitions are compiled into object code and then linked with the HI-MASS object library and any other required libraries (e.g., libg++) to form a single program (the executable model) as shown in Figure 7.

6 EXECUTING A MODEL

The user executes a model by simply running the program (executable model) with appropriate command line options. The command line options allow the user to specify the experimental frame (control state and variable initialization) files to use for the simulation run, to turn on run time trace information, and/or to specify single step event execution mode.

A simulation execution run has three main phases: model construction, model execution, and end of simulation output. During model construction the model objects (e.g., the AC's and MCS's) are constructed and initialized as specified in the experimental frame. The model then enters the simulation execution phase where the model is simulated until simulation termination conditions are satisfied. After the termination conditions are met, an end of simulation output phase allows the "Model", each AC, and each MCS to output end of simulation data and state information. This information includes the time of the last event executed in the model, the next event time for each AC, the current control state for each AC, and any data collection output specified by the modeler for an AC or a MCS.

HI-MASS provides trace information during the model construction and end of simulation output phases. A modeler can add additional trace information during each of these two phases if desired.

HI-MASS supports five types of run time trace information during the simulation execution phase. These include: (1) control flow (the flow of each AC's Point of Control over its HCFG), (2) intercomponent message traffic, (3) event list operations, (4) data associated with events, and (5) other data (e.g., data associated with edge condition evaluation, such as random variates returned by time delay functions). Control flow, message traffic, and event list trace information is automatically generated by HI-MASS. Data trace information, however, is model dependent, and thus must be explicitly specified by the modeler in the condition and event routines of the MCS's.

7 SUMMARY

We presented the two complementary types of specification structures (components and interconnections, and behavior specifications) that are used by HCFG Models. We described how a HIG is specified by constructing a set of CCS's using visual interactive modeling via the GUI supplied by HI-MASS, and how the behavior specifications (HCFG's) are constructed by specifying C++ code based on the object oriented foundation supplied by HI-MASS. We then presented overviews of the use of experimental frames, how the two types of specifications are transformed into an executable model, and how a model is executed in HI-MASS.

ACKNOWLEDGMENTS

HI-MASS was developed by the Simulation Research Group at Syracuse University under contract to the U.S. Air Force's Rome Laboratory.

REFERENCES

Fritz, D., T. Daum, and R. Sargent. 1995. *User's Manual for HI-MASS.* Simulation Research Group, 439 Link Hall, Syracuse University.

Fritz, D. and R. Sargent. 1995. An overview of control flow graph models. In *Proceedings of the 1995 Winter Simulation Conference.*

Hickey, R. 1995. Callbacks in C++ using template functors. *C++ Report* 7(2), 43–50.

Stroustrup, B. 1991. *The C++ Programming Language* (Second ed.). Addison-Wesley.

AUTHOR BIOGRAPHIES

DOUGLAS G. FRITZ is a graduate student at Syracuse University working towards a Ph.D. in Computer Engineering. His research area is hierarchical modeling for discrete event simulation. He received M.S. degrees in Electrical Engineering and Computer Science from Syracuse University and a B.S. degree in Electrical Engineering from The Pennsylvania State University. He was formerly with IBM as a development engineer for high speed switching systems.

ROBERT G. SARGENT is a Professor at Syracuse University. He received his education at the University of Michigan and has published widely. Dr. Sargent has served his profession in numerous ways and has been awarded the TIMS 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.*

THORSTEN DAUM is a graduate student at the University of Magdeburg who is working towards a degree in simulation and computer graphics. His interests include the development of Graphical User Interfaces for Visual Interactive Modeling. He was a visiting researcher with the Simulation Research Group and CASE Center at Syracuse University.