1996

# Hierarchical Control Flow Graph Models

Douglas G. Fritz
*Syracuse University, Simulation Research Group*

Robert G. Sargent
*Syracuse University, Simulation Research Group*

# HIERARCHICAL CONTROL FLOW GRAPH MODELS

Douglas G. Fritz
Robert G. Sargent

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

July 26, 1996

## ABSTRACT

Hierarchical Control Flow Graph Models define a modeling paradigm for discrete event simulation modeling based upon hierarchical extensions to Control Flow Graph Models. Conceptually, models consist of a set of encapsulated, concurrently operating model components that interact solely via message passing. The primary objectives of Hierarchical Control Flow Graph Models are: (1) to facilitate model development by making it easier to develop, maintain, and reuse models and model elements, and (2) to support the flexible and efficient execution of models. Hierarchical Control Flow Graph Models use two complementary types of hierarchical model specification structures, one to specify components and their interconnections, and the other to specify component behaviors.

## 1 INTRODUCTION

The Hierarchical Control Flow Graph (HCFG) Model paradigm is a hierarchical modeling paradigm for discrete event simulation that is based on and designed to be a hierarchical extension of Control Flow Graph (CFG) Models (Cota and Sargent 1990a). HCFG Models were designed to facilitate model development and support the flexible and efficient execution of models. Model development is supported through the use of hierarchy and encapsulation, and model execution is supported by algorithms that allow HCFG Models, based on CFG Models, to be executed on either sequential or parallel/distributed computer systems.

HCFG Models use two independent and complementary types of hierarchical model specification structures. One type of specification structure is used to specify a hierarchically organized set of encapsulated concurrently operating model components and the interconnections between those components. The other type of specification structure is used to specify the behaviors of the individual model components.

Discrete event simulation models based on the HCFG Model paradigm are easy to develop and provide (based on the underlying CFG Model representation) for flexible and efficient model execution on different computer architectures. The hierarchical modeling capability provided by the HCFG Model paradigm makes it easy to develop, use, maintain, and communicate models and model elements. The HCFG Model paradigm draws upon object oriented concepts such as encapsulation and inheritance (derivation) and lends itself to the development of generic and application specific libraries of reusable model elements. Given the existence of an appropriate set of model element libraries, HCFG Models can be constructed by simply "plugging" together existing model elements. The HCFG Models are also extensible, in that when an appropriate model element does not exist, a modeler can create a new model element, add this new model element to a model element library, and then use this new model element in the construction of the model.

HCFG Models support flexible and efficient execution via a set of CFG Model simulation execution algorithms that allow a model to be executed on either sequential or a parallel/distributed computer. A modeler using HCFG Models does not need to also be an expert on parallel/distributed computing in order to obtain efficient parallel/distributed simulation execution as is the case in some parallel discrete event simulation systems. No special or additional information is required from a modeler in order to execute HCFG or CFG Models on parallel/distributed computer systems as information required for efficient parallel/distributed simulation execution can be automatically extracted from the model by the simulation execution algorithms.

Most elements and relationships in an HCFG Model have a rather straightforward graphical representation that can be used as a basis for visual interactive modeling. The HCFG Model paradigm is computer language independent, and a simulation system based on HCFG Models may be implemented using any general purpose programming language. A prototype simulation system based on the HCFG Model paradigm was implemented using the C++ programming language (Fritz, Daum, and Sargent 1995; Fritz, Sargent, and Daum 1995).

The remainder of this paper is organized as follows. Section 2 gives a brief overview of CFG Models, HCFG Models and the high level operation of such models. Section 3 discusses the two types of model specification structures used in the specification of HCFG Models. Section 4 briefly introduces the use of "experimental frames" in HCFG Models, and in Section 5 a simple HCFG Model is presented to illustrate modeling using the HCFG Model paradigm. Finally, we summarize this paper in Section 6.

# 2 OVERVIEW

HCFG Models are based on hierarchical extensions to CFG Models. HCFG Models can be translated into equivalent CFG Models and executed on either sequential or parallel/distributed computer systems using any of the set of simulation execution algorithms developed for CFG Models. This section presents an overview of the CFG Model representation, the HCFG Model paradigm, and the operation of CFG and HCFG Models.

## 2.1 Control Flow Graph Models

Cota and Sargent (1990a) developed the CFG Model representation based on the modified process interaction world view (Cota and Sargent 1992). The primary objective of CFG Models was to make information useful for parallel/distributed simulation explicit in the model representation, thus enabling the development of a set of simulation execution algorithms for different types of computer architectures (Cota and Sargent 1990c). A modeler does not need to add any additional or special information to a CFG Model in order to efficiently execute a model on a parallel/distributed computer system. Conceptually, a CFG model consists of a set of independent, encapsulated, concurrently operating model components where each component has its own "thread of control" and the components interact with each other solely via message passing. The CFG Model representation is state based and favors an "active resource" view of modeling over an "active transaction" view. (GPSS (Schriber 1991) is a widely used system that favors the active transaction view.) Modeling from an active resource view means that the system is modeled from the point of view of the system's resources by describing the behaviors and interactions of those resources.

CFG Models use two complementary types of model specification structures. The first type of specification structure, called an Interconnection Graph, is used to specify the components that comprise the model and how those components are interconnected. The second type of specification structure, called a Control Flow Graph, is used to specify the behaviors of the individual model components. Each component in a CFG Model has an associated CFG behavior specification. A CFG defines the behavior of a specific *type* of component, and all components of the same type are specified via a single CFG. A CFG Model specification consists of one Interconnection Graph plus a set of CFG's (one CFG for each distinct type of component in the model).

An Interconnection Graph is a directed graph in which the nodes represent model components and the directed edges represent message channels that define a static routing pattern for intercomponent message traffic. Messages leave components through output ports and enter components through input ports. A component may have any number of input and/or output ports. Each channel connects one output port to one input port and each port is connected to exactly one channel (i.e., port connections are one-to-one). Each

channel generally carries only one type of message. This implies that there may be multiple channels between two components if those components need to communicate more than one type of message.

Intercomponent messages possess attributes that are used to carry information between components. The set of attributes possessed varies by message type, but all messages possess a "timestamp" attribute. The timestamp of a message is the time at which the message was sent. This is in contrast to the method generally used in parallel and distributed simulation in which a message's timestamp specifies the time at which the message is to be received and acted on by the message recipient.

Each component in a CFG Model has its own local simulation clock, and the value of each component's clock is strictly non-decreasing. (Operationally, components may only move forward through time.) Since the timestamp on each message is the time at which the message was sent (i.e., the value of the local clock of the sending component), the timestamps on the messages sent over each channel are non-decreasing. When a message is sent to a component's output port it is immediately (in zero simulation time) transported over the connecting channel to the corresponding input port. Messages sent over each channel arrive in the order in which they are sent. Each input port has an associated message queue, and messages arriving at an input port queue FIFO (First-In First-Out) in this message queue until the receiving component decides to receive and act on them; i.e., CFG Model components are "active" receivers (Cota and Sargent 1992). This is in contrast to object oriented simulation systems that generally use a passive receiver model in which messages are received and acted on by the receiving entity immediately upon their arrival.

The components in a CFG Model operate concurrently and are independent of each other except for message passing interaction. Each component has its own thread of control, its own local simulation clock, and its own set of local variables.

A Control Flow Graph is an augmented directed graph in which the nodes represent control states and the edges specify the set of possible control state transitions. A control state is a formalization of the "process reactivation point" (Cota and Sargent 1992).

Each component in a in a CFG Model has its own Point of Control (thread of control). Between events a component's Point of Control (POC) resides at a control state, and the control state where a component's POC resides at any point in simulation time is called the component's "current" control state. An event execution for a component consists of three distinct operations: (1) the component advances its local simulation clock to the time of its pending event, (2) the component's POC traverses an edge originating from the its current control state, and (3) the component carries out any additional actions specified by an event routine associated with the traversed edge. The control state that the component's POC arrives at following an edge traversal (event execution) then becomes the component's *new* current control state.

The "augmented" part of a CFG's augmented directed graph refers to a set of edge attributes. Each edge in a CFG, in addition to an originating and a terminating control state, has the following three attributes: a condition, a priority, and an event. The condition attribute specifies when (at what point in simulation time) an edge can be considered for traversal. The edge originating from the component's current control state whose condition is satisfied at the earliest point in simulation time is selected. Edge priorities are used to choose between edges whose conditions are satisfied at the same "earliest" time (i.e., to break time ties). Edge priorities must be unique among all edges originating from the same control state. An edge's event attribute specifies a set of actions to take (in addition to the local clock update and the POC edge traversal) as part of a component's event execution. These actions may include: receiving a message, sending one or more messages, and/or modifying the values of the component's local variables. Component behavior specification using CFG's is discussed in Subsection 3.2.

Cota and Sargent (1990c) developed a set of algorithms for the execution of CFG Models that allow CFG Models to be executed on either sequential computers or parallel/distributed computers. The parallel/distributed execution algorithms use information explicit in the CFG Model representation to automatically generate "lookahead" information (Cota and Sargent 1990b). The availability and quality of lookahead information is a key element in parallel/distributed simulation (Fujimoto 1990). Automatic generation of lookahead in CFG Models alleviates the need for a modeler to manually add such information to a model specification as is a common practice in parallel discrete event simulation.

The classes of simulation execution algorithms developed for CFG Models are shown in Figure 1. The sequential algorithms execute models on a sequential computer while the parallel/distributed algorithms execute models on parallel or distributed computers. The sequential-synchronous algorithm executes events

in a strict time order, whereas the sequential-asynchronous algorithm may reduce some simulation execution overhead (such as event list manipulations) by executing some events out of time order when the execution order of those events does not affect the simulation result. Conservative parallel/distributed algorithms only execute events when those events are guaranteed to be correct. (Conservative algorithms avoid deadlock through the use of either deadlock prevention or deadlock detection and correction.) Optimistic algorithms save model state prior to executing any event that is not guaranteed to be correct. If an optimistic algorithm later finds that an event execution was incorrect, it then "rolls back" to a previously saved state and continues execution from that "restored" state. The combined parallel/distributed algorithms attempt to execute in conservative mode whenever possible, but they may temporarily switch into optimistic mode when the executing processor would otherwise be idle.

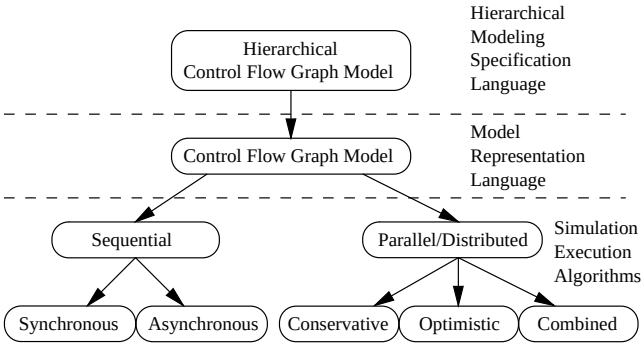



Figure 1: Modeling Language, Representation Language, and Algorithms

## 2.2 Hierarchical Control Flow Graph Models

HCFG Models are based on the modified process interaction world view (Cota and Sargent 1992) and were developed as a hierarchical modeling paradigm that can use CFG Models (Cota and Sargent 1990a) as a model representation language. Using CFG Models as a model representation language allows HCFG Models to be executed using any of the existing CFG Model execution algorithms.

While the CFG Model representation can be used for modeling, it was not designed for that purpose. The CFG Model representation is straightforward to use in the modeling of simple systems but models can become complex when modeling more complex systems. Also, CFG Model's provide only limited support for model element reuse. HCFG Models employ hierarchy and encapsulation for complexity management and provide a wider range of support for reuse than is provided by CFG Models. A modeler can develop a model using HCFG Models, transform the model into an equivalent CFG Model (Cota, Fritz, and Sargent 1994; Fritz and Sargent 1993), and then execute the model using any of the CFG algorithms shown in Figure 1. (Transforming an HCFG Model into its equivalent CFG Model is conceptually straightforward and computationally efficient.) HCFG Models are a superset of CFG Models in that any valid CFG Model is also a valid HCFG Model.

HCFG 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 a hierarchically organized set of encapsulated components that comprise the model and how those components are interconnected. The second type of specification structure, called a Hierarchical Control Flow Graph (HCFG), is used to specify the behaviors of the individual components of the model. A HIG is a hierarchical extension of an Interconnection Graph, and an HCFG is a hierarchical extension of a CFG. An HCFG Model specification consists of one HIG plus a set of HCFG's (one for each type of component in the model that requires a behavior specification).

HCFG Models use two distinct classes of model components: atomic and coupled. Atomic Components (AC's) correspond to the components used in CFG Models while coupled components have no counterpart in CFG Models. In HCFG Models only the AC's have HCFG behavior specifications.

Coupled components are encapsulated components that are formed by coupling together a set of atomic and/or coupled subcomponents. It is this recursive definition of coupled components that provides support for hierarchical component specification in HCFG Models. Each distinct type of coupled component in an HCFG Model is specified via a corresponding Coupled Component Specification (CCS). A modeler specifies the HIG for an HCFG Model by simply specifying the set of CCS's for the coupled components used in the model.

A CCS is a directed graph in which the nodes represent model components (atomic and/or coupled) and the edges represent the channels over which intercomponent messages flow. The channels in a CCS interconnect subcomponent ports in a manner analogous to a CFG Model Interconnection Graph. However, in a CCS, channels may also connect ports of the component's subcomponents to the "outside world" via a set of "external" ports that allow messages to flow through the enclosing coupled component's encapsulation boundary. Note that a CFG Model Interconnection Graph is simply a special case of a coupled component that: (1) has no connections to the outside world, and (2) in which all subcomponents are atomic. The specification of components and interconnections via CCS's is discussed in Subsection 3.1.

In HCFG Models each AC (Atomic Component) has a corresponding HCFG that defines the behavior for that type of AC. An HCFG is a hierarchical extension of a CFG in which the behavior of an AC can be recursively partitioned into a set of encapsulated disjoint "partial" behaviors that, when combined, define the AC's behavior. Behavior specification via recursive application of "divide and conquer" provides support for hierarchical behavior specification. This allows complex behaviors to be recursively broken down into sets of disjoint simpler behaviors, each of which can then be individually specified. Each (partial or total) behavior in an HCFG Model is specified using a behavior specification structure called a Macro Control State (MCS). A MCS is an extension of a CFG that provides support for hierarchical modeling and reuse at the sub-AC level. MCS's constitute the basic building blocks of an HCFG. A modeler constructs an HCFG largely by specifying a set of MCS's that, when combined, define the component's behavior. Component behavior specification using MCS's is discussed in Subsection 3.2.

Figure 2 illustrates the relationships between the various structures used in the specification of an HCFG Model. This model shows two coupled components, "a" and "c", and three AC's, "b", "d", and "e". Each coupled component is defined via a CCS (Coupled Component Specification) that specifies the subcomponents and interconnections that comprise the coupled component. Together, the CCS's for "a" and "c" completely specify the model's HIG. Coupled component "a", the top level component in the HIG, via transitive closure, encloses all other components in the model. Each of the AC's has an associated HCFG that specifies the behavior for that AC, and each HCFG is constructed from a hierarchically organized set of MCS's. The top level MCS of each AC encloses, via transitive closure, all other MCS's in that AC's HCFG in a manner analogous to the way the top level component of a model encloses all the components of a model. The HCFG for component "b" consists of a single MCS whereas the HCFG for component "e" is constructed using five MCS's. If the complexities of the MCS's used in components "b" and "e" are comparable, then the behavior of component "e" is likely to be (potentially five times) more complex than that of component "b".

The HIG and HCFG hierarchical structures used in the HCFG Model paradigm aid in the management of model complexity and make it easier to develop and maintain models. Also, the encapsulated model elements (CCS's and MCS's) used in HCFG Models provide support for model element reuse at both a higher level (CCS's) and at a lower level (MCS's) than is possible with CFG Models.

## 2.3 Model Operation

Conceptually, an HCFG Model consists of a set of independent, encapsulated, concurrently operating AC's that interact with each other via message passing. All intercomponent message traffic in an HCFG Model originates from and terminates at AC's. (Coupled components define the component hierarchy and the static routing pattern for the inter-AC messages, but they neither create nor destroy messages.)

The AC's in a model interact with a simulation executive that executes the model as specified by the simulation execution algorithm, and in this manner the simulation executive can be thought of as the simulation algorithm in operation. The simulation executive is a single central authority when executing a model using one of the sequential simulation algorithms, whereas it consists of a set of distributed interacting entities (executives) when executing a model using one of the parallel/distributed algorithms. When executing a model using one of the sequential algorithms the simulation executive has access to "global" information on

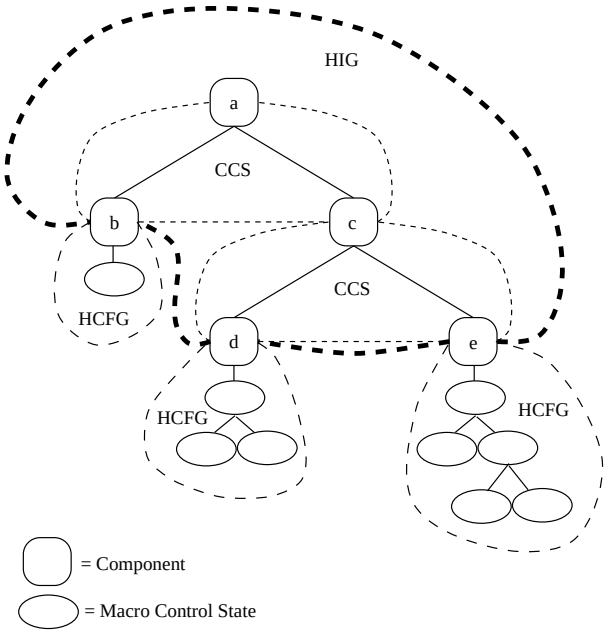



Figure 2: Relationships between Specification Structures

the state of the model and it directs the operation of all AC's in the model based on this information. When executing a model using one of the parallel/distributed algorithms each distributed executive is responsible for and directly interacts with only those AC's local to that executive. Distributed executives also interact with each other in the handling of operational details such as deadlock prevention.

The simulation executive (or distributed executive) extracts information from and issues operational directives to AC's and is responsible for the correct and efficient operation of the simulation model (Cota and Sargent 1990c). The executive is responsible for such activities as simulating the concurrent operation of AC's when executing a model on a sequential computer and for generating and using lookahead information when executing a model on a parallel/distributed computer system. The specifics of the interaction between the AC's and the simulation executive(s) differs based upon the simulation algorithm being used, but there exists a minimal level of interaction common across all the CFG Model simulation execution algorithms.

Conceptually an AC performs only two basic operations: (1) it selects an edge originating from its current control state, and (2) it executes its pending (next) event. The simulation executive uses information extracted from the AC's to determine when each AC should perform each operation and then directs the appropriate AC's to perform the appropriate operations in the appropriate sequence.

When an AC selects an edge as part of an edge selection operation, information associated with the selected edge becomes available to the simulation executive. The information associated with the selected edge includes the AC's next event time and whether the AC's next event is conditional or unconditional. A conditional event is an event that can be preempted (replaced with a different event) due to the arrival of a new intercomponent message. Also, each AC with a conditional next event informs the simulation executive upon receipt of any intercomponent message that might preempt the component's conditionally pending event.

When an AC is directed to execute its next event, the AC first advances its local simulation clock to the time of its next event. Then the AC's POC traverses the AC's selected edge and the AC carries out any additional actions specified by the event attribute (routine) associated with the traversed edge.

We present the synchronous sequential simulation execution algorithm as a concrete example of CFG Model execution. The sequential synchronous algorithm simulates CFG Models on a sequential computer and executes all events in strict time order. The sequential synchronous algorithm requires that each AC in the model have a unique AC priority. This AC priority is used by the simulation algorithm to break time ties between AC's. (Recall that CFG edge priorities were used to break time ties between edges within an AC.)

Cota and Sargent (1990c) developed an algorithm that analyzes a CFG Model and automatically assigns AC priorities in such a manner as to guarantee that the results of a sequential simulation of a model will be identical to the results of a parallel/distributed simulation of the same model.

An overview of the sequential synchronous simulation execution algorithm is shown in Figure 3. Construction of the model involves the construction and initialization (in computer memory) of objects representing the model. Model elements include model components, ports, interconnections, and component behaviors. Component behaviors include the CFG (or MCS) graph(s), variables, AC initial control state (POC location), and seeds for pseudo random number generators. Also, any initial intercomponent messages are created and placed in the appropriate input port message queues, and priorities are assigned to AC's for use in breaking time ties during model execution.

---

1. Construct and initialize the model; assign priorities to AC's.
2. Each AC performs an edge selection operation.
3. Repeat the following until termination conditions are met.
   (a) Select the AC with the earliest next event time, using AC priorities to break time ties.
   (b) The selected AC executes its pending event.
      i. The AC advances its local simulation clock to its next event time.
      ii. The AC's POC traverses its selected edge.
      iii. The AC carries out any additional actions specified by the event routine associated with the traversed edge.
   (c) The following AC's perform an edge selection operation.
      i. The AC that just executed an event.
      ii. Any other AC that both:
         A. had a conditionally pending edge, *and*
         B. received a new intercomponent message that could preempt the AC's conditionally pending edge.

Figure 3: Sequential Synchronous Algorithm

---

Each AC then performs an edge selection operation. Each selected edge has an associated time that determines the AC's next event time and whether the AC's next event is conditional or unconditional. One way of selecting the next AC to execute an event is to use a priority queue of AC's. If AC's are placed in a priority queue and selected based on the next event times of the AC's (using AC priorities to break time ties) then the AC with the earliest next event time (and highest priority in the case of time ties) will always be at the front of the priority queue. The sequential synchronous algorithm then simply selects the AC at the front of the priority queue and directs that AC to execute its pending event.

The selected AC then executes its pending event. The selected AC is then "flagged" to indicate that it must perform a new edge selection operation in order to determine its next event time. Also, any AC with a conditional next event that received an intercomponent message that could preempt its currently selected edge is also flagged for reevaluation. All such flagged AC's are removed from the priority queue, directed to perform a new edge selection operation, and then reinserted in the priority queue. Then, unless simulation termination conditions are met, the AC selection process begins again. This describes the basic operation of CFG and HCFG Models under the sequential synchronous simulation execution algorithm. The internal specification and operation of individual AC's is discussed in Subsection 3.2.

# 3 MODEL SPECIFICATION

An HCFG Model specification consists of one HIG plus a set of HCFG's (one per distinct type of AC in the model). The HIG specifies a hierarchically organized set of components that comprise a model and how those components are interconnected, while HCFG's describe the behaviors of the individual AC's in the model. Specification of a HIG, via a set of CCS's, is presented in Subsection 3.1, and the specification of HCFG's, via sets of MCS's, is discussed in Subsection 3.2.

## 3.1 Component and Interconnection Specification

The basic building block in an HCFG Model's HIG is the model component, and HCFG Models use two distinct classes of model components: atomic and coupled. Model components are encapsulated entities that have an external view and an internal view. From the external view, all model components (both atomic and coupled) have the following attributes: a name (instance name), a type (type name), a set of input ports, and a set of output ports. From the external view of a component it is impossible to distinguish coupled components from AC's. (The internal views of coupled components and AC's are covered in Subsections 3.1.1 and 3.2, respectively.) Each component in a model is an "instance" of a particular "type" of component. The distinction between instance and type is significant in that, if multiple model components are instances of the same type of component, then those components share the same type definition.

A component boundary is an encapsulation boundary. This means that the internals of a component are hidden from the component's external view, and conversely, the externals of a component are hidden from the component's internal view. The exception to this "hidden" rule is the set of ports through which messages enter or leave the component. Ports cross the component's encapsulation boundary and are visible from both the internal and external views of a component. Each port has the same identifier (name) on both sides of the component's encapsulation boundary and in this manner ports form the link between the internal and external views of a component.

The HCFG Model paradigm specifies model elements and relationships but does not dictate how these elements and relationships should be represented. In this paper we use graphical representations when we feel they more clearly convey information than a textual representation would. Some conventions we follow for the representation of model elements and relations are as follows. Components are represented via boxes, message channels are represented by line segments and their directions by arrows, and port identifiers (names) are located near the ports. Since a component's ports cross its encapsulation boundary and its port names are identical from both sides of the component boundary, in a graphical representation of a component its port names may be located either inside or outside the component as is convenient. Naming conventions we use are as follows. The first letter of a component type name is generally an uppercase letter whereas the first letter of an instance name is generally a lowercase letter. When instance and type names are shown together in a graphical representation, type names are distinguished by enclosing them in a set of parentheses "()".

An external view of a component is shown in Figure 4. This model component named "theBlueServer" is of component type "ExpServer". It has three input ports: "new–jobs", "suspend–operation", and "restart–suspended–job", and one output port: "completed–jobs". One possible definition of message types for such a component is as follows. (Note that intercomponent messages can carry information between components via message attributes.) Message arrivals on port "new–jobs" represent the arrival of a batch of new jobs requiring processing, and a message attribute "batch_size" specifies the number of jobs in each batch. A message arrival on "suspend–operation" indicates that the server should suspend operation until instructed otherwise, and a message arrival on port "restart–suspended–job" indicates that the server should resume operation, restarting any job that was in service at the time server operation was suspended. Each message departure on port "completed–jobs" represents a job that has completed service at "theBlueServer".

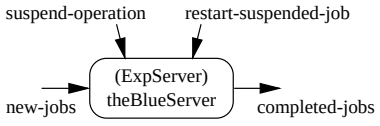



Figure 4: External View of a Component

### 3.1.1 Coupled Component Specification

Coupled components are encapsulated model components formed by coupling together other components. The internal view of a coupled component is the view from inside the component but outside all enclosed subcomponents. This internal view is specified via a "Coupled Component Specification (CCS)" that specifies: (1) a set of subcomponents that are coupled together to form the new coupled component type and, (2) how those subcomponents are interconnected.

Although coupled components do not have behavior specifications (HCFG's) like AC's do, they do have behaviors. The behavior of a coupled component is determined in an indirect manner by the behaviors and couplings of the subcomponents that comprise the coupled component. This derives from the fact that each coupled component encloses, directly and/or indirectly (recursion through the component hierarchy), one or more AC's. Note that coupled components that enclose identical sets of AC's may exhibit differences in behavior that are due solely to differences in the couplings (interconnections) of their enclosed subcomponents.

We next use a simple example to illustrate how a coupled component can be constructed from a coupling of subcomponents. The first step in forming a new coupled component from a set of subcomponents is to specify the set of subcomponents that are to be enclosed by the new coupled component and the interconnections (represented via channels) of those subcomponents. These interconnections include both connections between subcomponents and also connections between enclosed subcomponents and components external to the new coupled component. Suppose that we wish to construct a new coupled component type "C" that contains two components, "a1" and "a2", of type "A", and one component, "b", of type "B", interconnected as shown in Figure 5(a). (Note that the port identifiers (names) of the two instances of component type "A" are identical since they are of the same type.) The couplings (component interconnections) are represented graphically in the figure (e.g., output port "o2" of component "b" is connected to input port "in" of component "a2"). Ports "new–jobs" and "finished–jobs" of component "b" are intended to be connected to components that will reside outside the new coupled component and thus they are not connected in Figure 5(a).

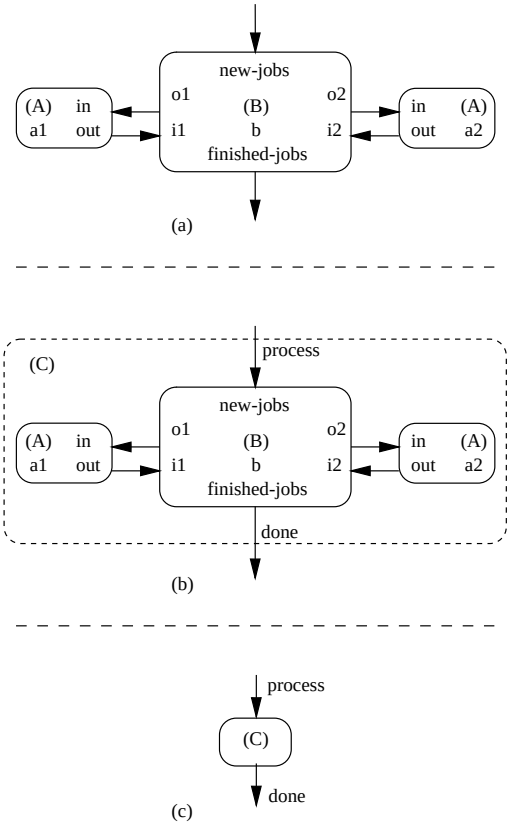



Figure 5: Coupling of Components

The next step in forming a new coupled component from a set of subcomponents is to draw an encapsulation boundary around the subcomponents that we wish to encapsulate, as shown by the "dashed" box in Figure 5(b). (We use a dashed box in Figure 5(b) to illustrate the drawing of a new encapsulation boundary for a component of type "C". All component boundaries in a CCS, including that of the enclosing coupled component, are normally shown as solid boxes.) Note that the encapsulation boundary should "cut" only those channels that will connect subcomponents contained within the new coupled component to components that will be located outside the new coupled component. Each channel cut by the encapsulation boundary forms an "external" port of the new coupled component and a unique identifier (name) must be assigned to each such port. Thus, the new coupled component type "C" (shown as "(C)" in Figure 5(b)) has two external ports, "process" and "done", through which subcomponents inside "C" may communicate with components outside "C". Note that Figure 5(b) is a graphical representation of the CCS that completely defines the new coupled component type "C". An external view of component type "C" is shown in Figure 5(c). Instances of this new component type "C" are encapsulated model components that can be used anywhere in a model that such a component is required.

The previous example illustrated a "bottom–up" approach of constructing coupled components by coupling together a set of existing subcomponents. Coupled components can also be constructed using a "top–down" approach. Using a top–down approach, a modeler first specifies the external view of the component (i.e., its type name and its ports) and then later specifies the internals of the component type. The top–down approach allows a modeler to use a component in the construction of a model while deferring specification of the component's internals to a later time when they can be addressed separately. A modeler can use either of these approaches or a combination of the two in the construction of an HCFG Model's HIG.

A general method for constructing an HCFG Model's HIG using a top–down development of coupled components is to recursively partition components into sets of interacting subcomponents until each of the remaining "non-partitioned" components has a behavior that can be easily specified via an HCFG. Each of these remaining non-partitioned components is then specified to be atomic and thus will have an associated HCFG behavior specification.

The AC's in a model operate concurrently and each AC has its own thread of control. Thus, if a particular model component has a "natural parallelism" in its behavior, then that component should be considered a candidate for partitioning into two or more AC's. Possible benefits of this partitioning include: (1) each of the new smaller AC's will likely have a simpler behavior and thus be easier to model via an HCFG, and (2) the maximum theoretical parallelism during simulation execution using a parallel/distributed algorithm increases with the number of AC's in the model.

### 3.1.2 Hierarchical Structures

A HIG is completely specified via a set of CCS's, each of which defines a coupled component type. However, to determine the set of components that comprise a model and the hierarchical organization of those components, one must construct the component hierarchy from the set of CCS's. This construction is accomplished by starting with the coupled component that encloses the entire model (the only component with no external ports) and recursively constructing each of that component's coupled subcomponents (as specified by the subcomponent's CCS).

Because a model's component hierarchy may not be obvious from the set of CCS's, it is desirable to have an auxiliary structure that shows a model's component hierarchy at a glance. HCFG Models use an auxiliary structure called a "HIG tree" for this purpose. A HIG tree is a rooted tree structure in which the nodes of the tree correspond to model components. Each node in the HIG tree has a pair of attributes that specify the corresponding component's instance name and type name. "Child" nodes of a node in the HIG tree correspond to the component's immediate subcomponents. The internal nodes of the HIG tree contain other components and thus correspond to coupled components, whereas the leaf nodes of the HIG tree correspond to AC's. A HIG tree is an abstraction of a HIG that captures the model components and their hierarchical relationships but not the component interconnections. A model's HIG tree can be constructed from its HIG (set of CCS's), but the reverse is not possible as the component interconnection information is not present in the HIG tree.

As stated above, each HCFG Model has one coupled component that encloses the entire model, and this component is the only component in a model that has no external ports. We refer to this component as

the "root" or "top level" component of the model as it corresponds to the root node of the HIG tree. Since components are encapsulated entities that interact solely via message passing, if a coupled component has no external ports, then those components inside this component are completely isolated from any components outside this component. If we can partition a set of components into two sets such that there are no interconnections (channels) between the two sets, then the two sets of components constitute at least two completely independent models.

HCFG Models also have a second type of auxiliary structure called a "HIG type tree". In a HIG type tree, the nodes of the rooted tree represent component "types" rather than component instances. A HIG type tree can be constructed from a HIG tree, but the reverse is not possible as the component instance name information is not present in the HIG type tree.

We use a simple example to illustrate the concepts of the HIG tree and HIG type tree. Suppose that we have an HCFG Model whose top level component type "M" is defined by the CCS in Figure 6. Assume that component types "A" and "C" are as shown in Figure 5 and also assume that component types "A" and "B" are atomic. The HIG for this model is completely specified by two CCS's (Coupled Component Specifications), one for the top level coupled component type "M", and one for the coupled component type "C". Component types "A" and "B" are atomic and thus have behavior specifications (HCFG's) rather than CCS's.

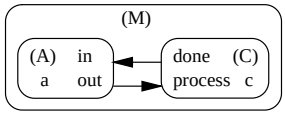



Figure 6: A Top Level CCS

The HIG tree for this simple model is shown in Figure 7(a) (component type names are shown in parentheses) and the corresponding HIG type tree is shown in Figure 7(b). Since all names in the HIG type tree are type names, there is no need to enclose the names in parentheses as is done in the HIG tree. The three vertical bars in Figure 7(b) next to the "(2)" indicate replication (i.e., more than one component of type "A" is contained within a component of type "C"). The "(2)" indicates that two components of type "A" are contained in a component of type "C" as subcomponents.

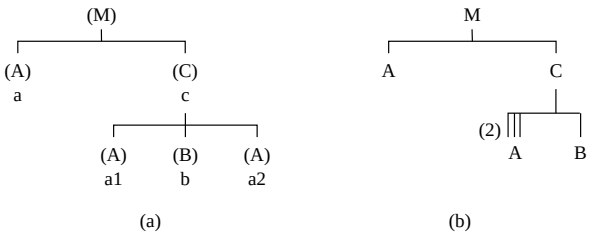



Figure 7: HIG Tree and HIG Type Tree

An HCFG Model's HIG specifies a hierarchically organized set of model components as defined by a set of CCS's. The HIG tree and HIG type tree are auxiliary structures that show at a glance the component hierarchy and component type hierarchy, respectively. The internal nodes of the auxiliary structures correspond to coupled components and the leaf nodes correspond to AC's. The specification of AC behaviors using HCFG's is discussed in the following subsection.

## 3.2 Atomic Component Behavior Specification

Each AC is an encapsulated entity with an external view and an internal view. From the external view (as discussed in Subsection 3.1) each type of component (coupled or atomic) has a type name, a set of input ports, and a set of output ports. From the internal view each type of AC has the same elements as from its external view plus an HCFG behavior specification.

The basic building block of an HCFG is the MCS. A MCS is an encapsulated behavior specification structure that is based on and designed to be an extension of a CFG. The MCS extensions to CFG's provide support for hierarchical behavior specification and model element reuse at the sub-AC level.

An HCFG behavior specification is a hierarchical behavior specification that is constructed from a set of hierarchically organized and interconnected MCS's. In addition to the set of MCS's, each HCFG also has a set of variables and functions that are referred to as "AC" variables and functions. These AC variables (e.g., the AC's local simulation clock) and functions are associated with the AC itself rather than any MCS. Any and all MCS's that constitute an AC's HCFG can access these AC variables and functions.

Internally, each MCS contains a set of MCS variables and functions that are "owned" by the individual MCS and are distinct from the AC variables and functions discussed above. Each MCS also possesses an augmented directed graph and a set of "handles" through which the MCS can access the AC variables, functions, and ports. If an HCFG consists of more than one MCS, then those MCS's also possess two additional elements. The first additional element is a set of "pins" in the MCS's encapsulation boundary through which the AC's POC (Point of Control) can enter or leave the MCS, and the second additional element is a set of handles that provide a MCS with access to variables and functions of other (ancestor) MCS's within the MCS hierarchy (of the HCFG). A MCS may only access information contained within an ancestor MCS if such access has been explicitly granted, and such access (via a handle) can only be granted by a parent to a child in the MCS hierarchy.

### 3.2.1 Fundamental Elements

In the case where an HCFG consists of a single MCS, that MCS is identical to a CFG. The fundamental elements that MCS's have in common with CFG's are discussed in this subsection, whereas the MCS extensions to CFG's that provide support for hierarchy and reuse are discussed in Subsection 3.2.3.

A CFG is a (control) state based behavior specification structure that is represented via an augmented directed graph. The nodes represent control states and the edges specify the set of possible control state transitions. Each AC has a POC (Point of Control) that moves from the AC's current control state, across an edge, to a new control state each time the AC executes an event. (Edges may originate and terminate on the same control state.)

Each edge in a CFG has an associated condition, priority, and event attribute. The condition specifies when an edge can be considered for traversal, edge priorities are used to to break time ties between edges, and an edge's event attribute (routine) specifies actions to be performed by the AC (in addition to the local clock update and the POC edge traversal) when the edge is traversed as part of an event execution.

An AC selects an edge based on the condition and priority attributes of the edges originating from its current control state. The evaluation of edge conditions may in turn depend on samples taken from distributions of random variables, the state of the AC's input port message queues, and/or the values of the AC and CFG (MCS) variables. Edge priorities must be unique for all edges originating from the same control state.

Each edge condition in a CFG belongs to one of the following three condition types: "time delay", "input port", or "boolean predicate". Edges are classified and edge conditions are evaluated based upon their edge type.

Edges with a time delay condition are called "TimeEdges", and associated with each TimeEdge is a time delay function whose evaluation returns a nonnegative real value $\Delta t$. Time delay functions may access local CFG and AC variables and may also sample from one or more random variable distributions. A TimeEdge's condition is satisfied (becomes **True**) with a local (AC) simulation time greater than or equal to $(t_{now} + \Delta t)$, where $t_{now}$ is the current value of the local simulation clock and $\Delta t$ is the value returned by the time delay function associated with the TimeEdge. A TimeEdge's time delay function is evaluated at most once between event executions. A TimeEdge retains the value returned by its associated time delay function until the AC executes its next event, after which the value is discarded, and thus a new value for $\Delta t$ must be generated the next time the TimeEdge is evaluated as part of an edge selection operation.

Edges with an input port condition are called "PortEdges", and associated with each PortEdge is an input port of the AC. Each PortEdge is associated with exactly one input port, however an input port may be associated with more than one PortEdge. A PortEdge's condition evaluation is determined by the status of the associated input port's message queue. If the input port's message queue is nonempty (i.e., there is at

least one unreceived message), then the PortEdge's condition is satisfied (**True**) with a local simulation time greater than or equal to the maximum of: (1) the timestamp on the first message in the input port's message queue, and (2) the current local simulation time ($t_{now}$). A PortEdge's condition is not satisfied (i.e., it is **False**) if the message queue of the associated input port contains no unreceived messages. Since the AC's in a model operate concurrently, intercomponent messages arrive asynchronously to the operation of each AC. The condition of PortEdges associated with empty input ports will immediately change from **False** to **True** upon the arrival of new intercomponent messages. Such a change in a PortEdge's condition in response to the arrival of a new intercomponent message may, in some cases, require that an AC redo an edge selection operation taking into account the updated status of the PortEdge.

Edges with a boolean predicate condition are called "BoolEdges", and associated with each BoolEdge is a boolean predicate that evaluates to either **True** or **False**. This boolean predicate can reference only variables contained within the CFG and the AC. (Recall, as described above, that AC and CFG variables, although both contained within an AC, are distinct.) A BoolEdge's condition is satisfied with a corresponding local simulation time greater than or equal to $t_{now}$ if the predicate evaluates to **True** at time $t_{now}$, otherwise the BoolEdge's condition is not satisfied (i.e., it is **False**). Since BoolEdge predicates are based on the values of AC and CFG variables and the values of those variables may only change during an event execution, a BoolEdge need be evaluated only once between event executions for the AC. We have found in behavior modeling using CFG's that BoolEdges with an "always **True**" predicate are used with sufficient frequency that we define a subtype of a BoolEdge, called a "TrueEdge", to be a BoolEdge whose predicate is defined to always be **True**.

Recall (Subsection 2.3) that an AC can perform only two basic operations: edge selection, and event execution. The simulation executive uses information extracted from the AC's to determine when each of these operations should be performed and then directs the appropriate AC's to perform the appropriate operations in the appropriate sequence. Two basic rules apply to the order in which the edge selection and event execution operations are performed for each AC. First, whenever an AC's POC arrives at a control state, the AC must successfully select an edge originating from that control state before that AC can execute its next event. Second, an AC with a conditional next event *must* perform another edge selection operation prior to executing its next event if it subsequently receives a message that could cause a PortEdge to preempt the AC's previously selected edge. Details of the edge selection and event execution operations are discussed below.

When the simulation executive directs an AC to reevaluate its next event information, the AC evaluates the conditions of the edges originating from the AC's current control state (based on their edge types) and selects the edge whose condition is satisfied at the earliest simulation time. If more than one edge originating from the AC's current control state has a condition that is satisfied at this "earliest" simulation time, then the edge priorities are used to select the highest priority edge whose condition is satisfied at this earliest time. The time associated with the selected edge becomes the AC's next event time.

PortEdges require special consideration during edge selection operations since only PortEdges have a condition that may change between between an AC's event executions. An AC may have PortEdges whose associated input ports have no unreceived messages (i.e., their message queues are empty), and evaluating the condition of any such PortEdge will, as described above, return **False**. If an AC has one or more such PortEdges originating from its current control state *and* the arrival of a new intercomponent message could cause such a PortEdge to be selected if the AC was directed to perform another edge selection operation, then the AC's pending event is a conditional event (i.e., it can be preempted by a different event). If, however, an AC's pending edge can not be preempted by any such PortEdge due to the arrival of a new intercomponent message, then the AC's pending event is unconditional.

An AC edge selection operation may fail to select an edge. If the simulation executive directs an AC to select an edge and no edge originating from the AC's current control state has an edge condition that is satisfied, then the edge selection operation fails and the AC has no pending edge. The simulation executive will never direct an AC to execute an event unless the AC has selected a pending edge with a finite next event time. An AC that fails an edge selection operation is considered to have a next event time of $+\infty$. It is common for an AC that is simply waiting for a message arrival to have a conditional next event time of $+\infty$.

When the simulation executive directs an AC to execute its pending event the AC performs the following

operations. First, the AC advances its local simulation clock to its next event time (the time associated with the AC's selected edge). Then, the AC's POC traverses the selected edge, and the AC carries out any additional actions specified by the traversed edge's event routine. The actions taken by an AC during an event execution may include updating the values of the AC's or MCS's local variables, sending one or more messages to one or more output ports of the AC, and/or receiving a message from one of the AC's input ports. An AC may receive a message from an input port only during those event executions in which the AC's POC traverses a port edge.

PortEdges also require special handling during event execution. If the edge traversed during an event execution is a PortEdge, the AC receives the first message from the PortEdge's associated input port message queue. The message is removed from the input port's message queue and the AC can examine and act on the contents of the message's attribute fields as part of its event action. Conceptually, the received message ceases to exist at the completion of the PortEdge's event routine.

If an AC requires no actions other than the clock update and POC traversal as part of an event execution, we say that the event associated with that edge is the "null event". The null event is commonly represented as "$e_{null}$" or "e–null()". A PortEdge with a null event (transparently to the modeler) receives and discards the message from its associated input port.

We use a graphical notation shown in Figure 8 to visually distinguish the different edge types used in CFG's. The condition and event attributes of an edge are located near the edge to which they belong. Edge priorities are indicated via a positive integer near the base (origin) of an edge. Lower numbers represent higher priorities, so an edge with priority one (1) indicates the highest priority edge originating from that control state. Edge priorities are only required when more than one edge originates from the same control state and no edge priorities are shown in Figure 8. Also, a TrueEdge (subtype of BoolEdge) is represented as a BoolEdge with a capital "**T**" near the edge's type symbol in lieu of a boolean predicate.

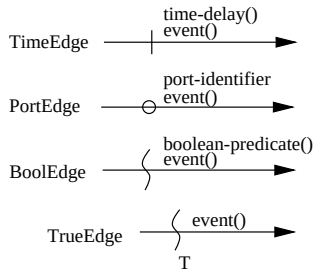



Figure 8: Edge Notation

### 3.2.2 A Two Class Server

In this subsection we demonstrate the behavior specification of a simple AC using a CFG (single MCS). We model the behavior of a simple two class server borrowed from Cota and Sargent (1990a). This two class server handles two classes of jobs using a "priority preempt/resume" job selection discipline. Each job is either a "high priority" job, or a "low priority" job. Jobs within each class are processed on a First Come First Serve (FCFS) basis. The server always works on a high priority job if one is available, and high priority jobs are always run to completion once they start service. If the server is busy with a low priority job when a high priority job arrives, the low priority job is preempted (work on it is suspended) and the server then begins working on the high priority job. The server then processes high priority jobs until all available high priority jobs have completed service. Work on any suspended low priority job is then resumed where it left off, followed by continued processing of any other low priority jobs.

Since the AC that we are modeling is a server that handles two classes of jobs, we assign this AC the type name: "2ClassServer". The external view of the two class server AC is shown in Figure 9. A "2ClassServer" AC has two input ports "hi-in" and "lo-in", and two output ports "hi-out" and "lo-out". In an AC of type "2ClassServer", each "job" arrival or departure is represented by a message. Thus a job arrival or departure is synonymous with (and represented via) a message arrival or departure, respectively. The priority of a

job arriving at the server is determined by the port on which it arrives. High priority jobs arrive on input port "hi-in" and low priority jobs arrive on input port "lo-in". As jobs finish service, they are sent out (as messages); high priority jobs on "hi-out" and low priority jobs on "lo-out".

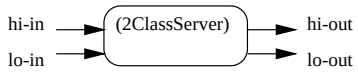



Figure 9: "2ClassServer" Type AC External View

We model the behavior of the "2ClassServer" AC using a CFG with four control states. We name these four control states: "I", "BL", "BH", and "BP" which stand for "Idle", "Busy-Low", "Busy-High", and "Busy-Preempt", respectively, as shown in Figure 10. (Note that this is just one of several possible ways in which to model this type of component using CFG's.) When the POC is at control state "I" the server is idle. When the control state is at "BL" the server is working on a low priority job (and no high priority jobs are available). When the POC is at "BH" or "BP" the server is working on a high priority job. If the POC is at "BP" there is a "suspended" low priority job upon which work will be "resumed" when there are no more high priority jobs to process. Three PortEdges and three TimeEdges show the possible control state transitions of the CFG.

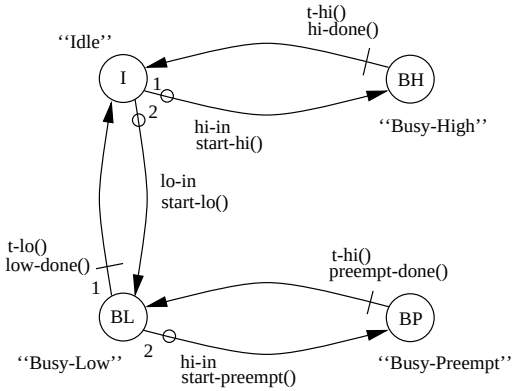



Figure 10: "2ClassServer" Control Flow Graph

When the POC arrives at control state "I", it will remain at control state "I" until a job is available. If a high priority job is available (there is an unreceived message on input port "hi-in") the POC will traverse the edge to "BH", executing the event "start-hi()" during the traversal. If a low priority job is available (there is an unreceived message on input port "lo-in") the POC will traverse the edge to "BL", executing the event "start-lo()" during the traversal. (Recall that a PortEdge's event routine receives a single message from its associated input port in addition to any other action it might take as part of an event execution.) If both types of jobs are available, the POC will move to "BH" because the edge to "BH" has an edge priority of "1" which is higher than the edge to "BL" which has a priority of "2". (Lower numbers indicate higher priorities. Also note that edge priorities do not need to be explicitly specified unless more than one edge originates from the same control state.)

When the POC arrives at "BL" from "I" it begins processing a low priority job, while when the POC arrives at "BL" from "BP" it resumes processing a previously preempted low priority job. If a high priority job arrives before the low priority job is completed, then work on the low priority job is suspended and the remaining time to completion for the job is saved by the "start-preempt()" event routine as the POC traverses the PortEdge to "BP". We assign a higher priority to the TimeEdge from "BL" to "I" than to the PortEdge from "BL" to "BP" so that in the case of a time tie (where a low priority job finishes service at the same time as a high priority job arrives) we send the completed low priority job on before we begin processing the new high priority job. This allows the low priority job to possibly continue processing in another component concurrently with the processing of the new high priority job in the "2ClassServer" component.

When the POC enters either "BH" or "BP", processing of a high priority job begins. Processing then continues for the duration specified by the time delay function "t-hi()" associated with the edges leaving "BH" and "BP". After the specified time delay for processing a high priority job the POC moves again. The event routines associated with the edges leaving "BH" and "BP" send a message to the "hi-out" output port indicating the completion of a high priority job. Even though the edges leaving "BH" and "BP" share the same time delay function, they have different event routines because the edge leaving "BP" has an additional responsibility to restore the state of the "preempted" low priority job that was saved by the "start-preempt()" event routine during the POC's traversal of the edge terminating on control state "BP".

Note that the definition of the "t-hi()" time delay function is straightforward as high priority jobs always run to completion once they begin processing, whereas the "t-lo()" time delay function on the TimeEdge from "BL" to "I" is slightly more complex because it must also take into account the processing time that a low priority job may have already received prior to and between preemptions by high priority jobs.

### 3.2.3 Support for Hierarchy and Reuse

The CFG representation is straightforward to use for modeling the behavior of simple AC's but CFG's can become complex when modeling more complex AC's. The number of control states and edges required to model a behavior may grow exponentially with the complexity of the behavior being modeled, and as the number of variables and functions required to model a behavior grows the CFG's namespace can become crowded. A crowded name space can be an inconvenience to a modeler and can lead to difficult to detect errors. (Consider a programming language in which all variables are global in scope.) Also, CFG's provide only limited support for model element reuse at the sub-AC level.

HCFG's provide support for hierarchical behavior specification and model element reuse at the sub-AC level. Using HCFG's, an AC's behavior can be recursively partitioned into sets of encapsulated disjoint partial behaviors, which when combined, completely specify an AC's behavior. Each partial behavior is defined via an associated encapsulated behavior specification structure called a MCS (Macro Control State). Each MCS is an independently specified and reusable model element. A set of hierarchically organized and interconnected MCS's form (along with a set of AC variables and functions) an HCFG behavior specification. This support for hierarchical modeling and encapsulation aids a modeler in managing model complexity. Generally, when an AC's behavior is partitioned into a set of disjoint partial behaviors, each of those partial behaviors tends to be less complex than the original behavior, and hence easier to model. Also, each partial behavior (MCS) has its own (less crowded) namespace. A modeler constructs an HCFG by specifying a hierarchically organized and interconnected set of MCS's along with a set of AC local variables and functions.

A MCS is an encapsulated behavior specification structure that has an external view and an internal view. From the external view a MCS has a name (instance name), a type (type name), a set of input pins, a set of output pins, and formal parameter list. From the internal view a MCS has a type, a set of input pins, a set of output pins, a formal parameter list, a set of variables and functions, an augmented directed graph, a set of handles to variables and functions of the AC, and a set of handles to variables and functions belonging to other (ancestor) MCS's within the HCFG.

Each MCS in an HCFG is interconnected to its parent MCS and to its immediate child MCS's in the HCFG hierarchy. These interconnections take two distinct forms, one form covers data and access sharing between MCS's, and the other form covers control flow interconnections that allow an AC's POC to flow between MCS's.

A MCS may grant one or more of its child MCS's access to variables or functions that are defined within the parent MCS or to any variables or functions that the parent MCS has been granted access to by its parent MCS. Access to variables and data may only be granted by a parent MCS to a child MCS and all such access permissions must be explicitly specified. MCS variables and handles (access) to external information are generally initialized via the MCS's parameter list and/or via an experimental frame. (The experimental frame concept and its use in HCFG Models is discussed in Section 4.)

The hierarchical modeling capability of HCFG's derives from the fact that MCS's may contain other MCS's. The nodes in a MCS's directed graph may contain both (simple) control states and/or other MCS's. Edges in a MCS's graph originate from and/or terminate on either control states or MCS pins. Edges that originate from control states are identical to the edges in CFG's, however, edges originating from MCS pins do not have the three (condition, priority, event) attributes that edges originating from control states have.

Also, each pin is the origin for *exactly* one edge.

The operation of an HCFG is an extension of the operation of a CFG. The POC for an AC resides at a control state called the current control state. This current control state is contained within a specific MCS, called the current MCS. Edges are selected in the same manner as in CFG's. The POC leaves the current control state over the selected edge and the action specified by the event routine associated with the traversed edge is carried out just as in CFG's. However, in an HCFG, the selected edge may terminate on either a control state within the same MCS, on an input pin of a child MCS, or on an output pin of the current MCS. If the selected edge terminates on a control state within the current MCS then the operation is identical to that of CFG's. If the selected edge terminates on an input pin of a child MCS, then the POC enters that child MCS through the input pin. If the selected edge terminates on an output pin of the current MCS, then the POC leaves the current MCS and enters the current MCS's parent MCS in the HCFG tree via the output pin.

When the POC traverses an edge that terminates on a pin, the POC continues to traverse a sequence of directed edges, starting with the edge leaving the pin, until it eventually arrives at a control state. In contrast to control states, which may have an arbitrary number of outbound edges, each pin has exactly one outbound edge, thus no edge selection algorithm is required for edges leaving pins. Since edges which originate from pins do not have the set of three attributes (priority, condition, and event) that edges originating from control states do, there is never a condition test required before traversing an edge originating from a pin, and there is no associated event to be executed during the traversal of an edge originating from a pin.

### 3.2.4 A Two Class Server Using MCS's

We show a variation of the "2ClassServer" example from Subsection 3.2.2 to demonstrate the use of MCS's for modeling partial behavior specifications. (The partitioning demonstrated in this example was selected for illustrative purposes rather than because it was particularly interesting, useful, or efficient.)

We encapsulate the processing of (non-preemptive) high priority jobs by drawing an encapsulation boundary around control state "BH" of our original CFG (Figure 10) as shown in Figure 11(a). This encapsulation boundary cuts the edges originating from and terminating on BH. Note than when a MCS encapsulation boundary cuts an edge, the condition, priority, and event attributes of the original edge remain on the side of the MCS boundary that contains the originating control state.

We replace the contents of the "BH" encapsulation boundary from Figure 11(a) with a child MCS named "delay1" as shown in Figure 11(b). (We represent MCS's contained within other MCS's graphically as ellipses so that they may be easily distinguished from (simple) control states.) The "delay1" MCS has two pins, which we label "in" and "out". These pins result from where the ("BH") MCS encapsulation boundary cut the two edges in the original CFG in Figure 11(a). The AC's POC may enter or exit the "delay1" MCS only via these pins. We next specify that the "delay1" MCS is to be a MCS of type "Delay1", and then define (specify the internal view of) a "Delay1" MCS type such that a "Delay1" type MCS models the behavior that was previously modeled by the elements within the "dashed" encapsulation boundary of Figure 11(a). We use the notation "`delay1` $\equiv$ `Delay1()`" to indicate that the "delay1" MCS in Figure 11(b) is of type "Delay1" and that a "Delay1" type MCS requires no parameters (discussed below).

A graphical representation of a "Delay1" MCS is shown in Figure 11(c). (The pins are represented as circles containing a "cross".) A "Delay1" MCS contains a single control state "S" and two edges. The TimeEdge originating from control state "S" has a time delay function "t-hi()" and an event routine "hi-done()" that perform that same functions as the originals by the same name did in Figure 11(a). The "t-hi()" time delay function and the "hi-done()" event routine are now associated with (part of) the "Delay1" MCS type definition shown in Figure 11(c).

Parameterization can significantly enhance the reusability of MCS's. We next demonstrate parameterization and reuse of MCS's. If, in addition to encapsulation the processing of (non-preemptive) high priority jobs, we also encapsulate the behavior of processing (preemptive) high priority jobs (with an encapsulation boundary around the "BP" control state of Figure 11(b)), we get a MCS as shown in Figure 12(a) with "Delay1" and "Delay2" type MCS's as shown in Figure 12(b). The "delay1" and "delay2" MCS's of Figure 12(a) would be specified as

```
delay1 ≡ Delay1()
```

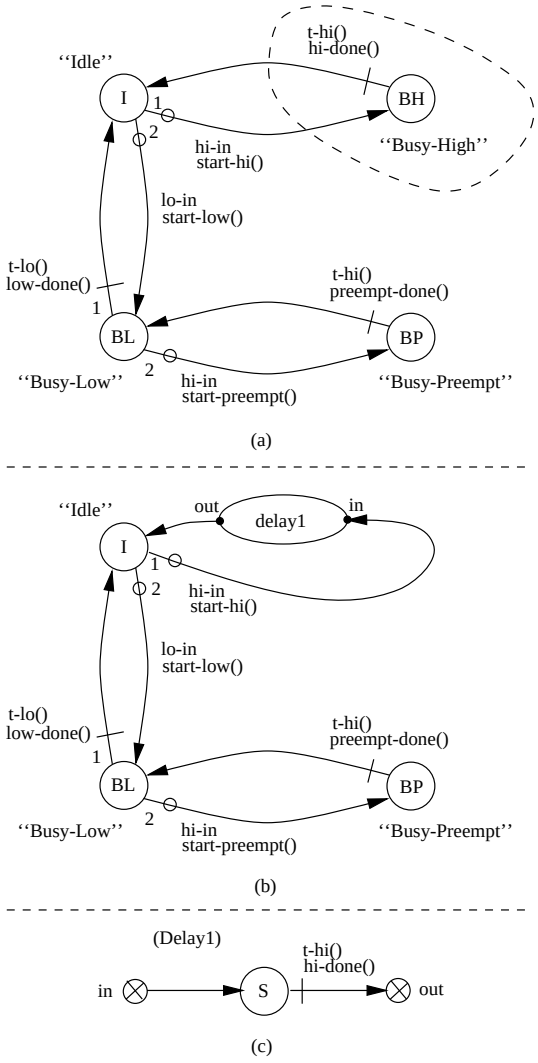



Figure 11: "2ClassServer" Having a Child MCS

```
delay2 ≡ Delay2()
```

Examining the "Delay1" and "Delay2" MCS types shown in Figure 12(b), we notice that the two types of MCS's differ only in their event routines ("hi-done()" versus "preempt-done()"). In this case it is trivial to create a single parameterized MCS type that can be used in place of the two distinct types. This new parameterized MCS type "Delay" is shown in Figure 12(c). The "Delay" MCS type has two parameters: a time delay function and an event routine. The "delay1" and "delay2" MCS's of Figure 12(a) can now be specified as follows.

```
delay1 ≡ Delay(t-hi,hi-done)
delay2 ≡ Delay(t-hi,preempt-done)
```

Now both "delay1" and "delay2" are instances of the same type of MCS that differ only in their event routines. (Note that the "Delay" MCS type has more flexibility than is required in the "2ClassServer" as it also allows for different time delay functions to be specified for the "delay1" and "delay2" MCS's.)

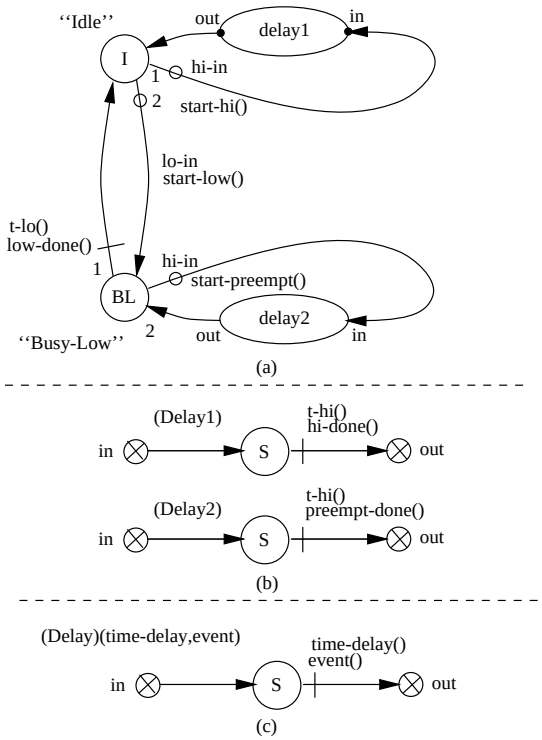



Figure 12: "2ClassServer" with Parameterized MCS Reuse

# 4 EXPERIMENTAL FRAME

HCFG Models support the use of experimental frames (Zeigler 1984). The "experimental frame" concept separates a model's definition from the set of model parameters used for a specific execution of the model. The experimental frame can be used to specify such information as: the initial control state (POC location) for each AC, the initial values for AC and MCS variables (including the seeds for random number generators), initial messages in input port message queues, desired data collection, model termination conditions, and, in some cases, even variations in the model structure (e.g., the number of servers for a simulation run). The experimental frame can also be used to specify parameters and handles during the instantiation of components and MCS's and can even modify HIG and HCFG hierarchies of the model.

Experimental frame information can be supplied to an HCFG Model either interactively or in batch mode. Information can be supplied via a user at a computer console, from a supervisory monitor program or artificial intelligence, or simply from a set of configuration files. Interactive specification of experimental frame information can allow for prompting, feedback, online help, and the use of default settings.

# 5 EXAMPLE

In this section we demonstrate modeling using HCFG Models by modeling a simple system. This simple system illustrates the various features of HCFG Models but does not demonstrate how HCFG Models can be used to model more complex systems; e.g., none of the AC's in this example have an HCFG that uses a MCS hierarchy of depth greater than two.

We model the behavior of a system that processes two classes of jobs using a two stage organization. The first stage consists of a single two class server as was described in Subsection 3.2.4, whereas the second stage of the system consists of two independent single servers, one for each class of jobs.

The HIG tree for the model we develop in this section is shown in Figure 13. From this figure it is easy to see the components that constitute the model and the hierarchical organization of those components.

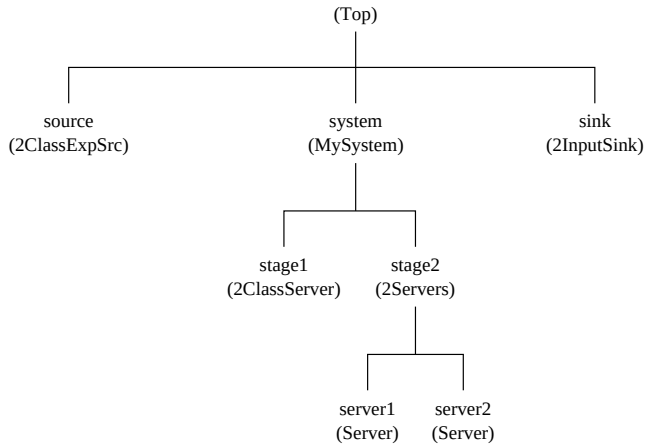



Figure 13: Model HIG Tree

We model the highest level of the component hierarchy (the top level CCS of the HIG) using three components as shown in Figure 14. The "source" component of type "2ClassExpSrc" models the arrival of new jobs requiring processing, the "system" component of type "MySystem" models the two stage processing system, and the "sink" component of type "2InputSink" removes jobs from the model after they have completed service. We model component types "2ClassExpSrc" and "2InputSink" as AC's, whereas we model component type "MySystem" as a coupled component, thus allowing it to be decomposed into simpler subcomponents.

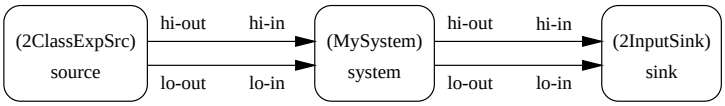



Figure 14: Top Level CCS

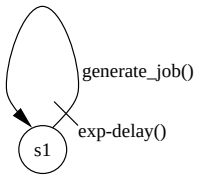
In this example we model each job as an intercomponent message (i.e., a message arrival to the "system" component corresponds to a job arrival). These intercomponent messages also have attribute fields that can be used to communicate additional information between components and also for any desired data collection activity.

We model job arrivals to the system (using component type "2ClassExpSrc") as having an exponential interarrival time, and furthermore, each arrival is randomly assigned a priority as it arrives. The mean interarrival time of jobs and the probability that a job arrival is a high priority job are two parameters that a modeler can specify via the experimental frame.

We model the behavior of the "2ClassExpSrc" type AC as shown in Figure 15. Whenever the AC's POC arrives at control State "s1" it will remain there for an amount of time determined by the "exp-delay()" time delay function. The "exp-delay()" time delay function is a function that samples from an exponential random variable with a mean value specified by the modeler in the experimental frame. When the POC traverses the self loop TimeEdge the AC executes the associated event routine "generate_job()". The "generate_job()" event routine (also shown in Figure 15) creates a new message, determines the job priority (based on a random variate), and sends the message (representing a job arrival) to the appropriate AC output port based upon the job's priority. This completes the description of the "2ClassExpSrc" type AC.

We next decompose the "system" component of type "MySystem" from Figure 14 into two processing stages as shown by the CCS in Figure 16. We model component "stage1" as an AC of type "2ClassServer" as described in Subsection 3.2.4, and we further decompose coupled component "stage2" of type "2Servers" into two subcomponents as shown by the CCS in Figure 17.

Both subcomponents, "server1" and "server2", are AC's of type "Server" (i.e. they share the same type definition). "Server1" processes only high priority jobs, while "Server2" processes only low priority jobs. The behavior specification of a "Server" type AC is shown in Figure 18. Recall that HCFG's (MCS's)




```
generate_job() {
    // create a new job (message)
    Message* newMsg = new Message();

    // sample from a 0 to 1 uniform distribution for priority_sample
    priority_sample = sample_from_0_1_uniform();

    if(priority_sample <= probability_that_job_is_high_priority) {
        // this is a high priority job
        // send job (message) to output port ``hi-out´´
        hi_out -> send(newMsg);
    } else {
        // this is a low priority job
        // send job (message) to output port ``lo-out´´
        lo_out -> send(newMsg);
    }
}
```

Figure 15: 2ClassExpSrc MCS and "generate_job()" Event Routine

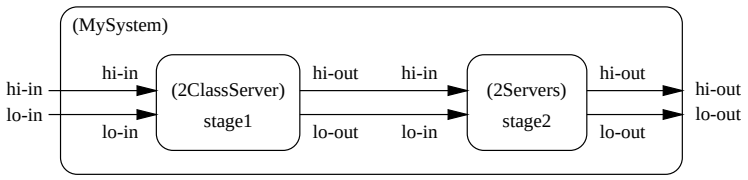



Figure 16: MySystem CCS

support parameterization, thus, even though "server1" and "server2" share a common behavior specification, a modeler can specify different mean service times for high priority and low priority jobs using the experimental frame. This demonstrates parameterized reuse of AC's in which parameters are specified via the experimental frame.

When the "Server" type AC's POC arrives at control state "I" (Idle) in Figure 18, the AC waits until a message arrives on input port "in" indicating that a job is available for processing. When a job is available the AC begins processing the job by executing the associated event routine "start-job()" and the POC moves to control state "B" (Busy). The AC's POC remains at "B" for an amount of time specified by the time delay function "delay(mean)" and then finishes processing of the job by executing the associated event routine "finish-job()". One of the tasks of event routine "finish-job()" is to send a message to the AC's output port "out" to signify the completion of the job. With the specification of an AC of type "Server" we have finished the specification of the component type "System" from Figure 14.

The only remaining component to be specified for our model is component "sink" of type "2InputSink" from Figure 14. We specify component type "2InputSink" to be an AC whose behavior is defined as in Figure 19. The "2InputSink" MCS has a single control state "s1" and two self looping PortEdges (one for each input port). When a message arrives on either input port of an AC of type "2InputSink" the appropriate edge is traversed. Note that the same event routine "remove-msg()" is associated with both

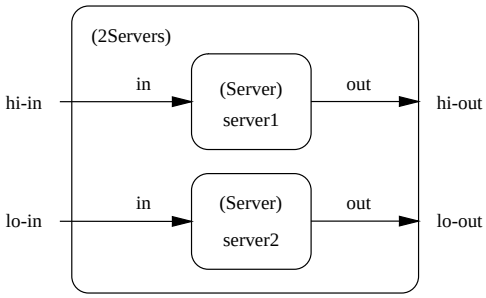



Figure 17: 2Servers CCS

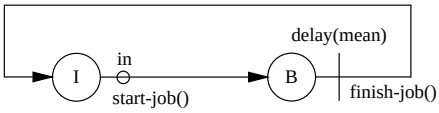



Figure 18: Server MCS

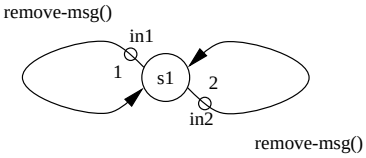



Figure 19: 2InputSink MCS

PortEdges. This event routine simply extracts any desired information from the received message and then destroys the message.

# 6 SUMMARY

We introduced the HCFG Model paradigm as a hierarchical modeling paradigm for discrete event simulation that makes it easy to develop and reuse models and model elements and supports the flexible and efficient execution of models on different types of computer architectures. We presented an overview of the specification and operation of CFG Models as the foundation upon which HCFG Models are based and then discussed the HCFG Model extensions to CFG Models. A high level overview of the operation of CFG and HCFG Models was then presented, using using the sequential synchronous simulation execution algorithm as an example. We then described the two types of complementary specification structures used in the construction of HCFG Models; how CCS's (Coupled Component Specifications) are used in the construction of a model's HIG (Hierarchical Interconnection Graph) and how MCS's (Macro Control States) are used in the construction of HCFG (Hierarchical Control Flow Graph) behavior specifications for the AC's (Atomic Components) of a model. We demonstrated hierarchical modeling and model element reuse in each of the two specification types. Lastly, we briefly introduced the experimental frame concept as supported by the HCFG Model paradigm and we then presented a complete model in order to illustrate modeling using the HCFG Model paradigm. The hierarchical modeling and reuse capability of HCFG Models makes it easy to develop, use, maintain, and communication models that can be efficiently executed on different types of computer systems. A modeler does not have to be an expert in parallel/distributed computing in order to achieve efficient parallel/distributed model execution of HCFG Models.

# References


Cota, B., D. Fritz, and R. Sargent (1994). Control flow graphs as a representation language. In J. Tew, S. Manivannan, D. Sadowski, and A. Seila (Eds.), *Proceedings of the 1994 Winter Simulation Conference*, pp. 555–559.

Cota, B. and R. Sargent (1990a, December). Control flow graphs: A method of model representation for parallel discrete event simulation. CASE Center Technical Report 9026, Syracuse University.

Cota, B. and R. Sargent (1990b). A framework for automatic lookahead computation in conservative distributed simulations. In D. Nicol (Ed.), *Distributed Simulation*, pp. 56–59. The Society for Computer Simulation.

Cota, B. and R. Sargent (1990c, November). Simulation algorithms for control flow graphs. CASE Center Technical Report 9023, Syracuse University.

Cota, B. and R. Sargent (1992, April). A modification of the process interaction world view. *ACM Transactions on Modeling and Computer Simulation 2*(2), 109–129.

Farr, S., A. Sisti, D. Fritz, and R. Sargent (1995). A simulation model of a surveillance radar data processing system using HI-MASS. In C. Alexopoulos, K. Kang, W. Lilegdon, and D. Goldsman (Eds.), *Proceedings of the 1995 Winter Simulation Conference*, pp. 1364–1370.

Fritz, D., T. Daum, and R. Sargent (1995, April). *User's Manual for HI-MASS.* Syracuse, NY: The Simulation Research Group at Syracuse University.

Fritz, D. and R. Sargent (1993, December). Hierarchical control flow graphs. CASE Center Technical Report 9323, Syracuse University.

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.), *Proceedings of the 1995 Winter Simulation Conference*, pp. 1347–1355.

Fritz, D., R. Sargent, and T. Daum (1995). An overview of HI-MASS (hierarchical modeling and simulation system. In C. Alexopoulos, K. Kang, W. Lilegdon, and D. Goldsman (Eds.), *Proceedings of the 1995 Winter Simulation Conference*, pp. 1356–1363.

Fujimoto, R. (1990, October). Parallel discrete event simulation. *Communications of the ACM 33*(10), 30–53.

Schriber, T. (1991). *An Introduction to Simulation Using GPSS/H.* John Wiley and Sons.

Zeigler, B. (1984). *Multifacetted Modelling and Discrete Event Simulation.* Academic Press.