Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

AN HCFG MODEL OF A TRAFFIC INTERSECTION SPECIFIED USING HiMASS-j

Thorsten Daum

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

ABSTRACT

The specification of a simulation model of a complex traffic intersection using the Hierarchical Modeling And Simulation System-Java (HiMASS-j) is presented. HiMASS-j is an object-oriented Java 1.1 based simulation software system that uses the Hierarchical Control Flow Graph (HCFG) Model paradigm. Models specified in this paradigm use two complementary hierarchical specification structures, one to specify the model components and their interconnections and the other to specify the behaviors of the individual components. Models are specified in HiMASS-j via visual interactive modeling.

1 INTRODUCTION

This is a companion paper to "A Java based System for Specifying Hierarchical Control Flow Graph Models" (Daum and Sargent 1997) contained in these proceedings. That paper provides a brief overview of the Hierarchical Control Flow Graph (HCFG) Model paradigm and discusses the portion of Hierarchical Modeling And Simulation System-Java (HiMASS-j) used for specifying HCFG Models via visual interactive modeling (VIM). It is assumed that the reader is familiar with that paper.

The purpose of this paper is to illustrate the use of HiMASS-j in the modeling of a non-trivial system. The system modeled is a complex multilane traffic intersection that involves vehicles, streetcars, and pedestrians. The purpose of this model is to study the relationships between traffic light patterns/sequences and the waiting time of vehicles in the intersection.

HCFG Models use two complementary types of hierarchical structures. The first type specifies the components that make up the model and how they are interconnected. This specification is called a Hierarchical Interconnection Graph (HIG). The second type of specification, the HCFG, is used to specify the behaviors of the individual atomic components of the model.

HiMASS-j is a Java (Arnold 1996) based object-oriented software system developed and tested on Sun SPARC workstations running Solaris and Pentium based personnel computers running Linux and Windows 95; however, due to the platform independent nature of Java, HiMASS-j can be run on any system that provides Java 1.1 support.

The remainder of the paper contains the following: an overview of the traffic intersection in Section 2, a description of the HIG, the HCFGs, and the Experimental Frame (EF) of the simulation model in Section 3, animation of the simulation model in Section 4, and a summary in Section 5.

2 THE SYSTEM

The simulation model created describes the busy intersection of Südring and Halberstädter Straße, which are two major streets in the city of Magdeburg, Germany (see Figure 1). The streets coming from the north, south, and west each have two vehicle lanes that expand into three lanes before the intersection. The street coming from the east remains at two lanes

Figure 1: The Intersection of Südring and Halberstädter Straße

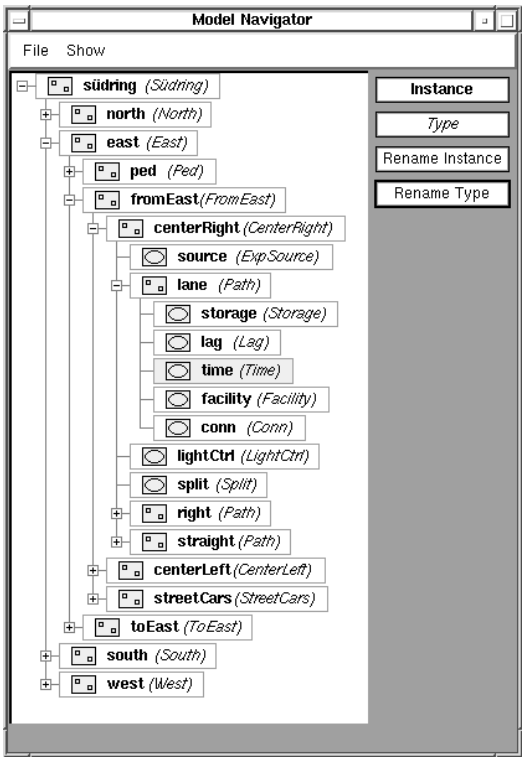

Figure 2: The Model Tree

for the intersection. Street car tracks are in the middle of the street between the vehicle lanes. Traffic consists of vehicles and pedestrians coming from all four directions and street cars coming from the north, west, and east. Traffic flow for vehicles, pedestrians, and streetcars is regulated by traffic lights. Traffic lights are synchronized and operate on fixed cycles preprogrammed for different times of the day. Currently, the intersection is not equipped to automatically adapt the traffic light cycle to varying traffic conditions. For a more detailed overview of the intersection see Figure 18.

Maps of the area and detailed data about the average hourly number of vehicles for each direction and time of day were provided by the city of Magdeburg Department of Transportation.

Streets coming from the north, west, and south respectively have individual lanes for vehicles turning right, going straight, and turning left. The street from the east has two vehicle lanes: one for vehicles turning right or going straight, the other for vehicles going straight or turning left. Street cars coming from the north turn right, street cars from the west can turn left or go straight, and street cars from the east go straight.

Traffic lights in the intersection work as follows: vehicles from the east have one traffic light, which means that vehicles turning right must yield to pedestrians and vehicles turning left must yield to pedestrians and opposing traffic. Traffic from the north, south, and west, respectively, has a traffic light for left turns and another traffic light for vehicles going straight or turning right. In general, all vehicles coming from one direction have a green light at the same time, with vehicles turning left usually having shorter green periods to accommodate pedestrian traffic. Street cars from the north turning right and from the west turning left have green lights when vehicles from the west turning left have a green light. Street cars from the west and east proceeding straight have green lights when vehicles from the east have a green light.

3- THE HCFG MODEL

The HIG of the intersection model has five levels. It contains over 400 AC instances and over 60 CC instances that were specified using 14 AC types and 20 CC types. The *complexity* of the intersection is modeled in the HIG; the ACs have simple HCFGs. Figure 2 shows part of the model tree in the Model Navigator window. Note that instance names start with a lower case letter and type names start with an upper case letter.

The remainder of this section discusses the HIG, a selected number of HCFGs, the specification of edge conditions and events, and the EF.

3.1- Hierarchical Interconnection Graph

The top level CC, Südring (the street name is commonly used for the intersection as well), consists of four CCs, north, south, east, and west (Figure 3). These CCs are connected by several multichannels specifying the interactions between the CCs. Each of these CCs contains three CCs; e.g., the East CC contains CC ped (specifying pedestrian traffic), CC

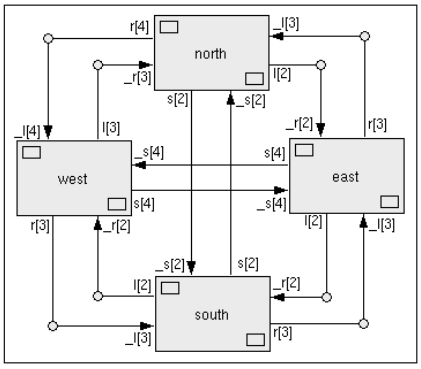

Figure 3: The Top Level CC

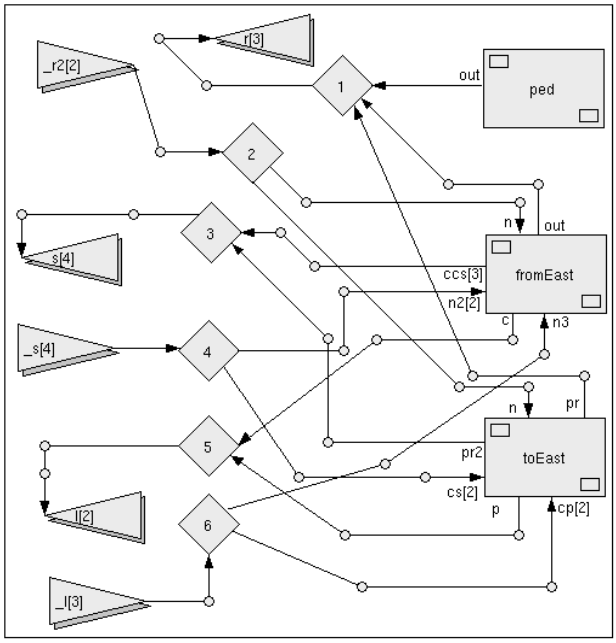

Figure 4: The East CC

fromEast, and CC toEast (see Figure 2). Channels, multichannels and six connection boxes are used to connect ped, fromEast, and toEast to the external ports of East (Figure 4). North, South, and West are specified accordingly. The FromEast CC (Figures 2 and 5) contains three CCs (centerRight, centerLeft, and streetCars) for specifying the two car lanes and the street car track, and one AC 'light'. Channels, multichannels, and two connection boxes are used to connect the components to each other and to the external ports of FromEast. The other From (e.g. FromWest) and To CCs are specified similarly.

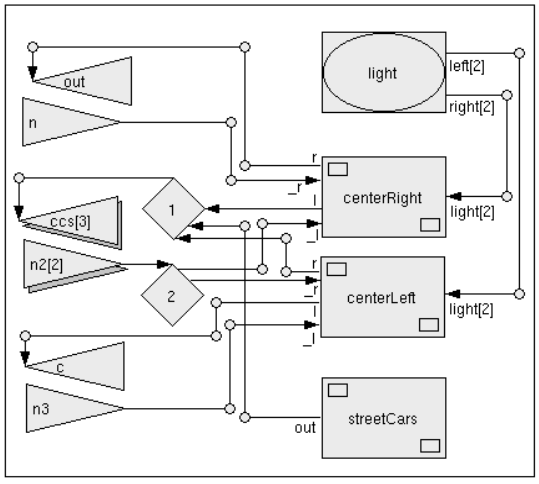

Figure 5: The FromEast CC

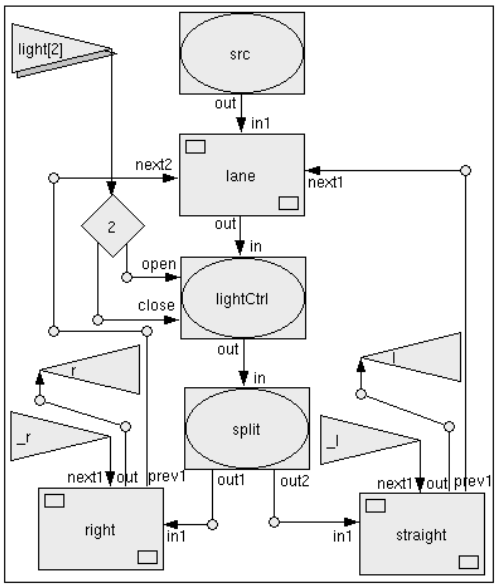

Figure 6: The CenterRight CC

The CenterRight CC (Figures 2 and 6) contains three ACs (source, lightCtrl, and split) and three CCs (lane, right, and straight) which are of type Path (Figures 2 and 7).

A Path CCS specifies the most basic element of an intersection, which is a continuous stretch of road, track, or sidewalk where traffic can always flow unrestricted, i.e., there are no intersecting traffic or other obstacles. A Path contains five ACs. In the model, Path instances connect to other instances of Path, traffic light controllers, sources, sinks, etc.

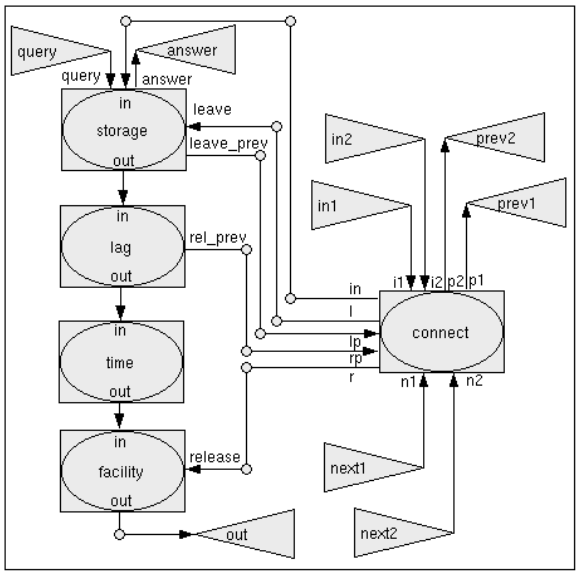

Figure 7: The Path CC

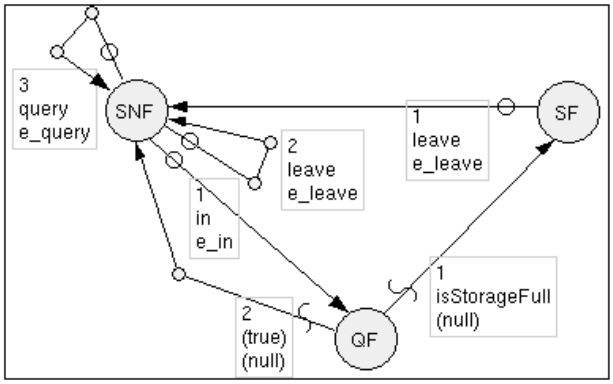

Figure 8: The Storage HCFG

Every instance of Path requires three parameters: a distinct name, a capacity, and a time. The capacity is the maximum number of entities (vehicles, streetcars, or pedestrians) that the Path can accommodate, time is the fastest possible time in which an entity can travel the entire length of the Path, and the name is for the generation of trace information. Both time and capacity relate to the length and the speed of a Path instance and to the length properties of the kind of entities on this Path.

Although it has a simple interface, parameterized instances of the same Path CC are used to specify all roads, walks, and tracks in the model. Components used to model the other properties of the intersection are TrafficLight, LightCtrl, Split, Block, ExpSource, and Sink.

Two kinds of messages are used in the model. "Entity messages" represent vehicles, street cars, and pedestrians. Other messages solely change and/or query the state of components, such as the state of traffic lights, the number of entity messages in a Path instance, etc.

CC Path has five input and four output ports. "Entity messages" can enter Path through input ports in1 and in2 and exit through output port 'out'. Path has two 'in' ports because channels can connect the 'out' ports of two Path instances to in1 and in2 of another Path instance, representing, e.g., vehicles merging into the same lane. Output ports prev1 and prev2 transmit messages to the respective previous instances of Path to indicate that an entity message has entered this Path instance and thus has left the Path instance it came from. Input ports next1 and next2 connect to ports prev1 and prev2 of the next Path to receive such messages. Input port 'query' receives messages that query whether the storage is empty and output port 'answer' carries a return message if this is the case.

3.2- Hierarchical Control Flow Graphs

Some of the HCFGs contained in CC Path and the HCFGs of AC ExpSource, AC TrafficLight and AC LightCtrl are described.

3.2.1- The Storage HCFG

The Storage HCFG (Figure 8) contains three Control States (CSs). The initial CS is SNF, which represents the state of the Storage when it is not full. This state indicates that entities can enter this path. Entities entering a path are modeled by a port edge associated with input port 'in' originating at SNF. Whenever SNF has the Point of Control (POC) and there is a message waiting at port 'in', the edge from SNF to QF is traversed executing event e_in because this edge has the highest priority. The Event e_in receives a message, increments the storage counter, sends a copy of the received "entity message" to output port 'out' (Figure 7) to be processed by the next AC of Path, and sends another message to output port leave_prev that will eventually cause the Storage AC in the previous path to decrement its counter.

Two edges originate from CS QF (Queries if storage is Full). Both edges have *null events* (events that do nothing) and serve the sole purpose of traversing the POC to CS SF (Storage Full) or back to SNF, based on the value of the storage counter. The boolean condition isStorageFull is always evaluated first because the associated edge has the highest priority. If the condition is true, the POC traverses to SF; otherwise, the lower priority edge, which by definition as a TrueEdge is always true, will cause the POC to traverse to SNF. If a message (indicating that an entity has left this path) is waiting at input port 'leave', the edge is selected by the POC to traverse to SNF from either SF or SNF and event e_leave will be executed, decrementing the storage counter. A message waiting at input port 'query' indicates that another path, that may be potentially blocked by entities traveling on this path, wants to know if this path is empty. The event e_query will send a message to output port 'answer' only if the storage counter is zero.

3.2.2- The Facility HCFG

The Facility HCFG (Figure 9) has two CSs: RE(leased) and SE(ized). A PortEdge associated with input port 'in' originates at RE. If an "entity message" is waiting at 'in' and RE has the POC, the message is received, copied to output port 'out' and the POC traverses to SE. If a message is waiting at input port 'release' and SE has the POC, the message is received, deleted, and the POC traverses back to

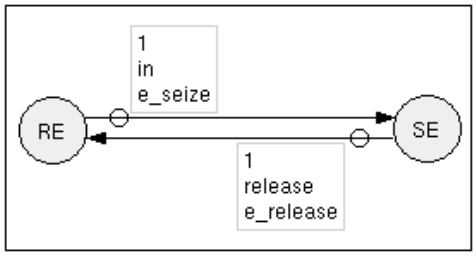

Figure 9: The Facility HCFG

RE so that the next "entity message" at 'in' can be received, etc.

3.2.3 The Connect HCFG

The Connect HCFG (see Figure 7) simply routes messages between its ports. Its purpose is to reduce the number of ports in CC Path and to simplify the Path's subcomponents; in particular Storage, Lag, and Facility. Messages coming in on next1 and next2, e.g., can be routed to Storage.leaves or Facility.release. Without the routing provided by Connect, the number of these ports would have to be doubled and additional edges added to the Storage, Lag, and Facility ACs.

3.2.4 The ExpSource HCFG

ExpSource generates "entity messages" with exponential interarrival times. The mean and the type of entities to be generated are AC parameters, and the mean and the seed of the random number generator (RNG) can be changed through the Experimental Frame. ExpSource (Figure 10) contains one CS S1 and one MCS ExpDelay. Although ExpSource is simple, encapsulating the exponential delay has a purpose. The implementation of the RNG and the desired distribution can be encapsulated, hiding the implementation details. Should the RNG or distribution need to be changed, a new delay MCS can

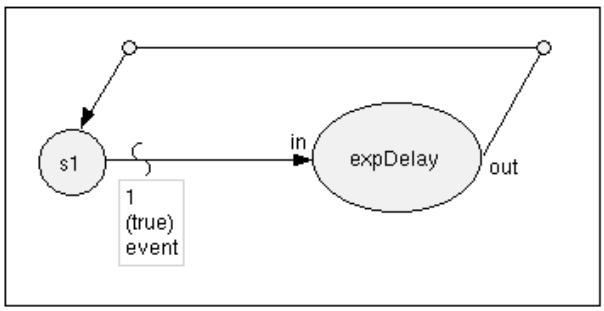

Figure 10: The ExpSource HCFG

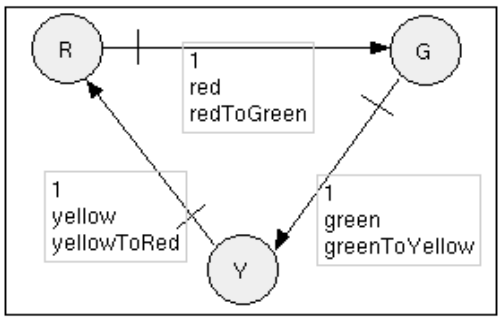

Figure 11: The TrafficLight HCFG

easily be plugged in without changing the functionality of ExpSource, e.g., by using the EF. Furthermore, ExpDelay itself can be reused in different models as exponential delays are very common and the specification is independent of the ExpSource MCS.

3.2.5 The TrafficLight and LightCtrl HCFGs

The TrafficLight HCFG (Figure 11) contains three CSs R(ed), Y(ellow), and G(reen) and three TimeEdges that connect R to G, G to Y, and Y to R. The time delay functions red, yellow, and green return the times the light is in a particular state. The values for red and green are specified in the EF. The events redToGreen and yellowToRed send messages to output ports open and close, respectively, causing the connected LightCtrl to change its state accordingly.

The LightCtrl HCFG (Figure 12) contains two CSs: OP(en) and CL(osed). Two edges originate from OP: (i) a self-looping PortEdge associated with input port 'in'; its event copies the message from 'in' to output port 'out' and (ii) a PortEdge to CL associated with input port 'close'. This port is connected to TrafficLight.close, which sends a message when the light changes to red causing the POC to traverse the edge to CL, until a message to 'open' causes the POC to traverse the PortEdge originating at CL back to OP. In the HIG, a LightCtrl instance is placed between two Path instances to block "entity message" traffic when the light is red.

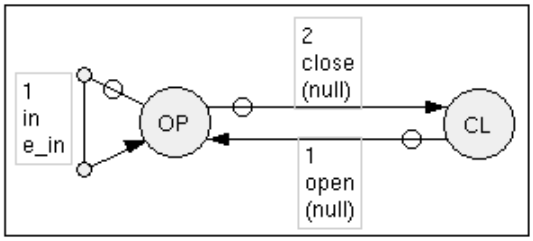

Figure 12: The LightCtrl HCFG

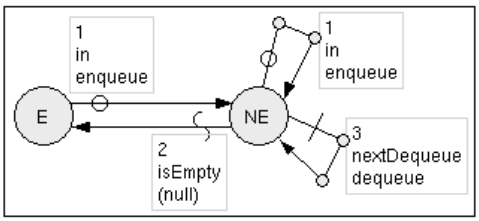

Figure 13: The Lag HCFG

3.3- Specifying Conditions and Events

In HiMASS-j time delay functions, boolean functions, and event routines are specified in dialog boxes by text and helper functions. A basic knowledge of the Java syntax is required.

The Lag AC (Figure 7) is used to illustrate the specifications of time delay and boolean functions and event routines. Lag AC is a simple server advancing the timestamp of each incoming "entity message" by the time it takes an entity to travel one entity length. If that time has passed, it copies the message to output port 'out' and sends a new message to output port 'rel_prev' that will eventually cause the Facility in the previous path to change its state to released. This is necessary to prevent entities that are queued up at a stop to start moving simultaneously after the stop condition such as a red traffic light has been changed.

The Lag HCFG (Figure 13) has two CSs, two events enqueue and dequeue, one time delay function nextDequeue, and one boolean function isEmpty. The two CSs are N(ot)E(mpty) and E(mpty) indicating whether there are message or no messages in the internal queue of Lag. When a message is sent to input port 'in', the PortEdge originating at the CS that has the POC becomes true and the event enqueue is executed. This event is specified by selecting the Edit tool and clicking on "enqueue" in the edge attribute label. This will cause the HiMASS-j system to open an Event Editor dialog box. A modeler then specifies the event by typing it into the text box. Helper functions aid in the specification of the event. Figure 14 shows the fully specified event. Line 1 of the event was produced by the helper function Receive

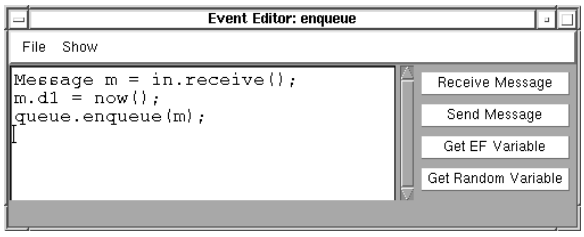

Figure 14: The enqueue Event

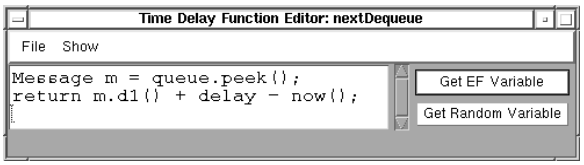

Figure 15: The nextDequeue Time Delay Function

Message. It declares a local variable m of type Message, which is a standard HiMASS-j data type, and initializes m with a reference to a Message object that is returned by the receive method of input port 'in'. In line 2 the current time of the AC is stored in field d1 of the message. The standard HiMASS-j messages have two integer, two double precision floating point, and two string member fields. (A modeler can, if desired, create other types of messages). In line 3 the message m is added to the message queue "queue" of the Lag MCS. A message queue is a commonly used data structure and is provided by HiMASS-j.

After event enqueue is executed, the POC traverses to CS NE. If there are no messages waiting at input port 'in', the BoolEdge from NE to E is evaluated. Since a message has just been added to the message queue, the edge is false and the TimeEdge originating at NE is evaluated. The TimeEdge becomes true after a simulation time delay specified by the time delay function nextDequeue. A time delay function returns a non-negative value (including zero, which indicates that the condition is true immediately). The time delay function is specified by clicking on "nextDequeue" in the edge attribute label, which will open the Time Delay Function Editor dialog box. Figure 15 shows the Editor box with the specified time delay function. The code in line 1 obtains a reference to the first message in the queue without removing it from the queue. In line 2 the time the first queue element must still remain in the queue is computed by adding the fixed delay to the time the message was added to the queue and subtracting the current time of this AC, and then returned.

If the TimeEdge becomes true, event dequeue (Figure 16) is executed. The code in line 1 removes the first element from the queue and assigns it to a local

Figure 16: The dequeue Event

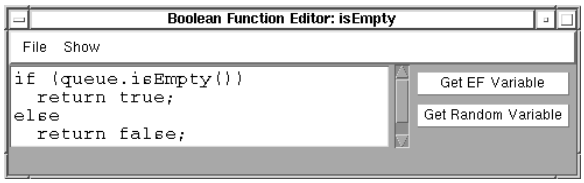

Figure 17: The isEmpty Boolean Function

variable m of type Message. In line 2 a new message m2 is created and initialized with the current time. (This is the standard method of creating messages). The integer field i2 of m2 is then assigned with the value in m.i2 which is the id of the previous path. In line 4 m is send to output port 'out' and in line 5 m2 is send to output port rel_prev, notifying the previous path that the entity represented by m has left it.

After the execution of dequeue the POC traverses the edge back to CS NE. If no new message is waiting at 'in', the BoolEdge to CS E is evaluated. Figure 17 shows the specification of isEmpty, the boolean function of the edge. Boolean functions are specified similarly to time delay functions. The only difference is that time delay functions return a non-negative number whereas boolean functions return either true or false. If the message queue of HCFG Lag is empty (Line 1), the function returns true and the POC traverses the edge to CS E. If the queue is not empty, the function returns false (Line 4) and the next edge originating at NE is evaluated.

Time delay functions, boolean functions, and events can be reused and additional helper functions can be added to HiMASS-j, making it easier to specify functions and events.

3.4- Experimental Frame

HiMASS-j makes the use of Experimental Frames straightforward and easy. Dialog boxes that provide the functionality to specify model element parameters, variables, initial CSs, and types each have a checkbox "get from EF". By selecting this checkbox, a modeler makes a parameter, variable, etc. available to the EF. The HiMASS-j system automatically generates an entry in an EF file that can be edited by using dialog boxes and helper functions. During model initialization, the HiMASS-j simulator attempts to initialize the element with a value from the specified EF file. If no matching entry is found, a default value is used, and if no default value is specified in the model, the simulation run terminates with an error message.

The intersection model makes extensive use of the EF. The means and seeds of Random Number Generators for instances of ExpSource and Split are specified in the EF, as well as the return values of the time delay functions of TrafficLight instances. That makes changing the average number of entities (e.g., for a different time of day) and the traffic light cycle easy.

4- ANIMATION

A simulation run produces a trace file that can be used for the animation of the model using Proof Animation (Wolverine Software 1995). Proof Animation is a vector-based, file-driven, post-processing animation system. It has a CAD-like drawing ability for creating layouts, paths, and shapes and provides a set of animation commands and the ability to process program-generated sequences of those commands. Figure 18 shows a picture of the animation.

The Proof layout and paths were generated manipulating a computerized (scanned) image of a map of the intersection provided by the city of Magdeburg Department of Transportation. Paths in Proof are sets of ordered, directional line and/or arc segments. If an entity is placed on a path it will move along the path at the specified speed until it reaches the end of the path or until it is blocked by another entity, in which case it will be temporarily halted until the blockage goes away. That is the desired behavior for the traffic intersection.

There is a direct relationship between Proof paths and Path CCs. A Path CC models a part of the roadway where moving entities can only be blocked by entities in front of it (that may be stopped at a traffic light). The time and capacity parameters of the Path CCs were obtained from the length and time attributes of the corresponding Proof paths. A Path instance adds a Proof command to the trace file each time an entity message enters the Path instance. This command is used by Proof Animation to place an entity on the specified Proof path, removing it from the previous Proof path and thus providing for smooth animation.

5- SUMMARY

A description of a complex HCFG model using HiMASS-j was presented. HCFG models contain a HIG specifying the model components and their interconnections and a set of HCFGs specifying the behavior of atomic model components. The HIG and HCFGs are specified via VIM, using graphical user interfaces and dialog boxes. Events and edge conditions are specified by text adhering to Java syntax. Dialog boxes and helper functions aid this process significantly. No advanced programming knowledge is required.

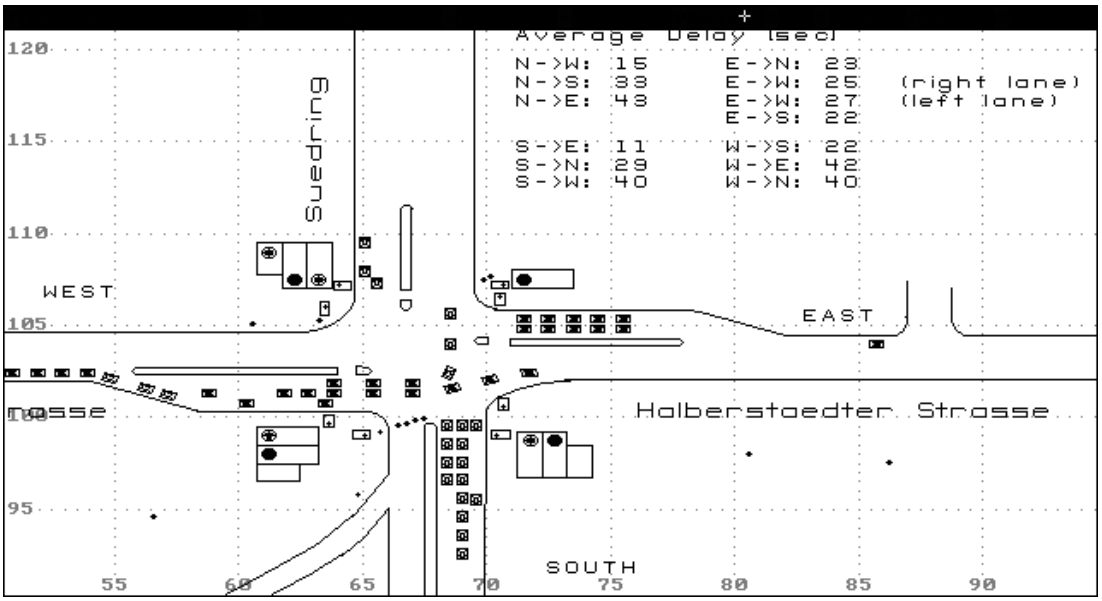

Figure 18: Animation of the Intersection

HiMASS-j offers a flexible way to build discrete event simulation models. The hierarchical nature of the HCFG Model paradigm allows for the representation of complex systems in a way that is intuitive and comprehensible. The intersection model uses over 400 ACs, yet could be structured in a way that was clear and straightforward; the HIG consists of five levels of hierarchy which contain the *complexity* of the model.

HiMASS-j has comprehensive capabilities for reuse that simplified the building of the described model. Although over 400 AC instances were employed, only 14 AC types were used and thus only 14 ACs had to be specified. Furthermore, 60 instances of the CC type Path were used. The ability to parameterize model element instances was crucial for effective reuse.

The EF features provided by HiMASS-j are useful. Initial conditions and other parameters can easily be changed between simulation runs without the need to recompile the model. This allows a compiled model to be given to a user who needs no knowledge of the way the model was specified to conduct meaningful simulation runs.

VIM as provided by HiMASS-j proved to be an intuitive tool to build discrete event simulation models easily and quickly. The visual aspect of VIM provides a natural way of modeling, the modeler is less likely to build a model that is incorrect, thus a verified model can be achieved faster than with text based tools for model specification. VIM is also useful if the simulation results should be animated because the visual structure of the HIG can be specified to resemble the layout of the animation.

ACKNOWLEDGMENTS

Professor Robert G. Sargent of Syracuse University supported and aided in the writing of this paper. Uwe Ilgenstein of Otto von Guericke University developed the Proof Animation layout for the intersection model.

REFERENCES

Arnold, K. and J. Gosling. 1996 *The Java Programming Language*. Reading, Mass.: Addison Wesley.

Daum, T. and R. Sargent. 1997. A Java based system for specifying hierarchical control flow graph models. In: S. Andratdottir, K. Healy, D. Withers, and B. Nelson, eds., *Proc. of the 1997 Winter Simulation Conference*.

Wolverine Software. 1995. *Using Proof Animation (Second Edition)*. Annandale, Va.: Wolverine Software Corporation.

AUTHOR BIOGRAPHY

THORSTEN DAUM is a graduate student at Otto von Guericke University in Magdeburg who is working towards a degree in simulation and computer graphics. His interests include the development of visual interactive modeling systems for simulation and Java software. He is a visiting researcher with the Simulation Research Group and CASE Center at Syracuse University.