An Object Oriented Simulator for Conceptual Graphs

Kiran S. Sastry

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Dr. Walling R. Cyre, Chair
Dr. Robert P. Broadwater
Dr. Richard E. Nance

May 10, 2001
Blacksburg, Virginia

Keywords: Conceptual Modeling, Simulation, Conceptual Graphs, CGIF

An Object Oriented Simulator for Conceptual Graphs

Kiran S. Sastry

(ABSTRACT)

This thesis deals with the design and implementation of an object-oriented simulator for conceptual graphs. Conceptual graphs are a means of representing information and knowledge. In particular, they may be used to represent the behavior of mechanisms. Conceptual graph simulation provides the means for verifying that the conceptual graph model of the system is a proper representation of the mechanism. The motivation for the design of this simulator is to help a conceptual graph model designer overcome the imprecision and ambiguity inherent in the English language. When a person translates an English language specification of a system to a conceptual graph model, the model may be incomplete, owing to semantic gaps in the English language specification.

The simulator attempts to help the designer fill in these gaps by pointing out missing concepts and relations needed to simulate the model. This thesis covers the issues involved in designing such a simulator, and the implementation of the simulator in Java. The working of the simulator is demonstrated by simulating sample conceptual graphs. Also, a set of action procedures, and a small library of device schema graphs are created, so that devices may be effectively modeled.

This work was funded in part by the National Science Foundation, Grant MIP-9707317.

Acknowledgments

I would like to thank Dr. Walling Cyre for making this thesis possible. His earlier work in the area of simulating conceptual graphs has proved an invaluable guide during the course of my thesis. His readiness to discuss design issues and resolve problems has proved to be immense help. I take this opportunity to thank Dr. Broadwater and Dr. Nance for serving as members of my committee.

I would like to sincerely thank Finnegan Southey, University of Waterloo, for the Notio API he has provided to the conceptual graph research community. His readiness to answer questions and clarify doubts has helped me in my thesis.

I take this opportunity to thank Dr. Harry S. Delugach, University of Alabama in Huntsville, for the prototype conceptual graph editor he has designed. The editor proved of immense use in designing test cases to test the simulator.

I definitely owe a note of thanks to the National Science Foundation for providing the financial support for this project.

I am also indebted to my parents and the rest of my family for all the support they provided me. Last, but not the least, I thank all my friends and all others who have directly or indirectly helped me in my efforts.

Table of contents

List of Figures

Chapter 1 Introduction

Conceptual graphs have been used to represent a variety of knowledge, including knowledge that describes behavior of people, animate beings, displays and mechanisms. Conceptual graphs have been implemented in a variety of projects for information retrieval, database design, expert systems, and natural language processing. This chapter looks briefly at the issue of conceptual graph simulation with the help of example conceptual graphs. The importance of being able to simulate conceptual graphs is discussed, and the motivation for the work is stated. In addition, an outline of how the rest of the document has been organized is presented.

1.1 Overview of conceptual graph simulation

When systems are modeled using conceptual graphs, the resulting conceptual graphs should be validated in order to ensure that they are accurate representations of the actual systems. In simple conceptual graphs, it is relatively easy to mentally trace possible states and activities of the system. This allows the modeler to decide if the conceptual graph correctly models the behavior of the system.

When the modeler needs to validate complex conceptual graphs or hierarchical graphs (conceptual graphs that contain sub-graphs), a mental validation of the system's states and activities may not be practical. In such situations, the validation of the conceptual graph should be carried out using simulation. Thus simulation presents itself as a means of validating the model of a system.

An example conceptual graph that describes the multiplication of two numbers is shown in Figure 1-1. In this graph, the processor multiplies six by five and stores the product by overwriting the zero.

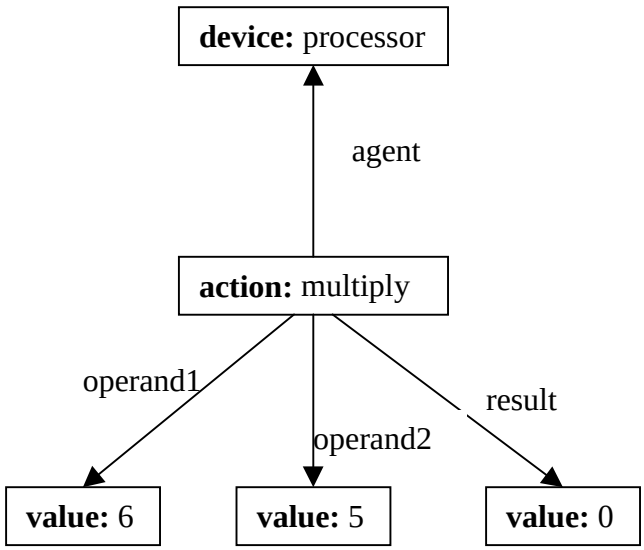

Figure 1-1–A simple conceptual graph.

As an example of conceptual graph simulation, consider the graph of Figure 1-1. Suppose one wishes to simulate this graph to observe the behavior. From the graph of Figure 1-1, we may develop the following conclusions. First, the action word *multiply* must be associated with some underlying procedure or type definition that knows what types of entities may be related to it. Second, this procedure should also know how the execution of the *multiply* action affects these related entities. Let us see what this means with respect the example conceptual graph in Figure 1-1. Executing a *multiply* action should read the two operands from the graphs, perform the multiplication of these two numbers, and then store the product of the *multiply* operation into the result value.

In this discussion, the action word *multiply* was assumed to have a procedure defined for it–possibly stored in a library of concept procedures. Such a library–one that contains procedures for each verb in the English language–can easily get out of hand. So, it is desirable to have a type definition for the action word *multiply* in terms of a modest set of primitive actions that read, copy, write, combine or operate on values of attributes. In terms of this set of primitive actions, the *multiply* action would have two read operations (for reading the two operands), one operate to perform the multiply, and then one write to store the product.

Another example of a conceptual graph is seen in Figure 1-2, which models the English sentence “The processor takes 20 nanoseconds to fetch the instruction from the memory into the instruction register.” Executing the conceptual graph in Figure 1-2 calls an action procedure associated with the word *fetch*. This procedure then checks the graph for information about what to fetch, where to fetch it from, where to store it after it is fetched, and how long the operation takes. Once the procedure has determined that it has sufficient data to perform the action, it simulates the agent (processor) fetching the operand (instruction2) from the source (memory) to the destination (instruction register) after the delay (20 time units), causing a change in the value of the result (instruction2). In terms of the set of primitive operations listed above, a *fetch* action would primarily be a single copy action, copying the operand to the result.

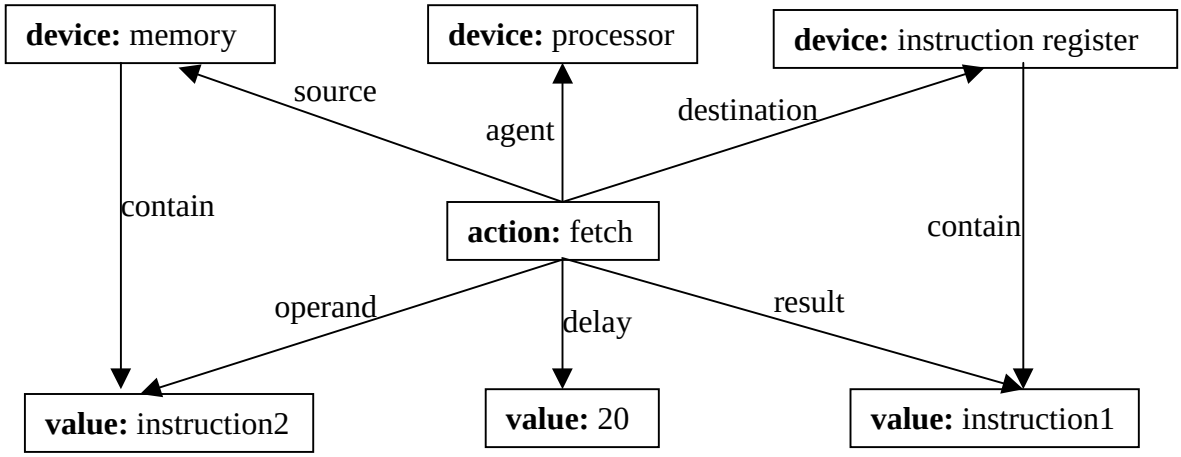

Figure 1-2–Another example conceptual graph.

Figures 1-1 and 1-2 present a useful introduction to conceptual graph simulation, as we have drawn two important conclusions from the above discussion. Firstly, we need each action word to have an action procedure. Secondly, we need to ensure that each action procedure can be built from a smaller set of operations such as move, copy, replace, read and write. Another conclusion may be drawn from Figure 1-2 above–though it is not immediately seen. Information like operand and result are important for simulation, in their absence, the action word will not know what data is to be fetched, or where it is to be stored, and hence cannot simulate the graph. This necessary information

comes from the *operand* and *result* relations between the action concept for fetch and the value concepts for instruction1 and instruction2.

1.2 Motivation for the work

The advantages of performing simulation in order to validate a model are twofold. Firstly, the results of simulating a model provide information about the accuracy of the model. An inaccurate model may be refined before being subject to re-validation. This iterative process of refinement and simulation may be continued until the behavior of the model matches the predicted behavior of the system under stipulated operating conditions.

Secondly, when the accuracy of the model has been established, we may use the model (instead of the actual system) in order to predict the behavior of the system when it operates under non-stipulated conditions. Using the model instead of the actual system may prove to be helpful, especially when systems such as nuclear reactors or missile-deployment systems are being modeled.

The ability to simulate conceptual graph models of systems is important in the field of artificial intelligence. This document is a part of an ongoing effort to automate the design process of digital systems. The design process is automated by developing software applications that take in an English description of a digital system and generate Hardware Description Language (HDL) code for the system.

For this process of design automation, conceptual graphs are being used as an intermediate stage for representing the information contained in the English description of the system. This document involves the design and implementation of a simulator for the conceptual graphs, so that the conceptual graphs can be simulated and verified to be an accurate representation of the information contained in the English description. This step is of vital importance as it is very essential to verify the validity of the conceptual graph before beginning the process of generating HDL code from the conceptual graph.

1.3 Contribution of the present author

Cyre [Cyre, 1998] has implemented a prototype of the conceptual graph simulator using Prolog. This prototype is an event-driven simulator for conceptual graphs. The prototype only simulated event and simple action concepts. The Prolog simulator is an implementation of the algorithm by Cyre. It uses Prolog files for the simulation engine and the different conceptual graphs. Since all the source and test files are just Prolog files, this simulator is easy to implement and modify.

But the Prolog prototype presents two main drawbacks. The first drawback is that it is written in Prolog. The Prolog simulator is not very efficient when it has to simulate large conceptual graphs. Implementing the simulator in an object-oriented manner will help both in adding new features and promoting code reuse. Secondly, the simulator expects the input graphs to be in the format specified by Cyre. However, this format is not in widespread use, and is not a standard. Hence the user has to perform an additional step of converting the graph format before it can be simulated.

The principal contribution of the present author was in overcoming the above two drawbacks. The present author used Java™, an object-oriented programming language, for the implementation. The present author implemented the algorithm proposed in [Cyre, 1998], providing a command line interface to the simulator, with normal and debug modes of operation. He also created a small library of action procedures and device schema graphs that are used by the simulator. The simulator is also tested on substantial graphs that were manually created from technical documents such as patents.

The present author also used the Conceptual Graph Interchange Format (CGIF) for textual representation of the conceptual graphs [Sowa, 1999]. Though CGIF is not yet a standard, a working draft of the proposed ISO standard is available at [Sowa, 2001a]. Also, an external software package, the Notio API, was adapted for implementing the simulator. The Notio API [Southey, 2000] provides a set of implementation-independent classes, as well as a means for dealing with conceptual graphs in CGIF.

1.4 Outline of the document

Chapter 2 reviews related work by other researchers, both in the fields of conceptual modeling and discrete event simulation techniques. How a combination of the two areas may be used to implement a conceptual graph simulator is discussed.

Chapters 3 and 4 examine the design for a conceptual graph simulator. Chapter 3 takes a look at the high level design. Those components of conceptual graphs that can be simulated are examined. The implementation of graph simulation using queues is discussed in this chapter. Chapter 4 deals with the low-level design of the conceptual graph simulator. This chapter deals with all the details involved in the design. The various steps performed by the simulator are detailed here, and the necessity of performing such steps is explained.

The implementation of the conceptual graph simulator using Java is discussed in Chapter 5. Software modeling tools such as Unified Modeling Language (UML) are used to describe the software organization of the simulator. The standards adopted for representing conceptual graphs is explained. The Notio API that has been used to build the simulator is also briefly explained. The testing of the simulator, and the results obtained, are explained in Chapter 6. The domain of interest for the purpose of simulation is discussed. The test setup used, and the methods for future testing and verification are also outlined.

Chapter 7 deals with conclusions and future work. It talks about the current status of the conceptual graph simulator, in terms of the number of action procedures it currently supports and the user interface. This chapter also suggests some future enhancements to the simulator. The appendices include notes to users (Appendix A), and to future developers (Appendix B). Appendix C lists pseudo-code for some of the action procedures. A UML report for the software architecture is also attached in Appendix D.

Chapter 2 Related Work

There has been an increasing amount of research in the fields of conceptual modeling and simulation. Discrete Event Simulation (DES) is another interesting topic of research, as most of the present-day digital systems. This chapter reviews some relevant work in the above fields. It discusses the conclusions reached by different authors, and states how their conclusions prove useful to the design of the conceptual graph simulator.

2.1 Conceptual Graphs

Before we look at traditional conceptual modeling and simulation techniques, it is necessary to have a brief introduction to conceptual graphs. In [Sowa, 1984], Sowa states "Conceptual graphs form a knowledge representation language…" Sowa defines a conceptual graph as a "finite, connected, bipartite graph." A conceptual graph is finite, as its size is limited by the amount of storage that the human brain or a computer can allocate to it. It is connected because a disjoint set of nodes would simply constitute multiple connected graphs. The graph is bipartite; it may be partitioned into two types of nodes. The arcs within the graph connect different types of nodes; there are no arcs that connect similar types of nodes.

Sowa classifies the two types of nodes of a conceptual graph as concepts and conceptual relations. Concept nodes describe entities and relation nodes may be used to connect concepts. Every conceptual relation has one or more arcs, which must be linked to concepts. Every conceptual relation is classified by the number of arcs incident to it. For example, a conceptual relation with one arc is called a *monadic* relation, one with two arcs is called a *dyadic* relation. Finally, a single concept may form a complete graph in itself, but a single conceptual relation cannot. The relation must have at least one arc, which is linked to a concept.

When a conceptual graph is represented, each concept of the graph is usually represented by its *type label*. Two concepts are of the same *type* if they have the same type label. The difference between type labels and concepts is that each concept is an instance of a *type*, and the set of all such instances is represented by a unique label called the *type label*. The conceptual relation between concepts and types is the "is-a" relation. For example, when we talk about a certain computer, we say, "*adrg.ece.vt.edu* is a computer." This "is-a" relation defines the *type* of a concept. Similarly, we may say, "a computer is a device." *Type hierarchy* is defined as "a partial ordering over the set of type labels." By the use of type hierarchy, *subtypes* and *super-types* may be defined. From the above example, *computer* is a subtype of *device* and *device* is a super-type of *computer*.

As discussed earlier, a concept is an instance of a type. An *individual marker* is used to identify a concept. A particular concept is completely represented in a graph by its type label and an individual marker. Alternatively, the *generic marker* * is used to represent a *generic concept*, a concept that does not refer to any particular concept but to just an instance of that type. The *referent* of a concept is defined to be either an individual marker or the generic marker. The referent of a concept may either be a simple label, or it may another conceptual graph that describes the concept. In the latter case, the referent of the graph is called the *descriptor* of the concept. A *line of identity* is used to link two concepts that refer to the same marker. The *line of identity* is defined as "a connected, undirected, graph whose nodes are concepts, and whose arcs are pairs of concepts, called *coreferent links*." Two concepts are said to be *coreferent* if they share a line of identity. Two conceptual relations are said to be *duplicate* if they are of the same type each arc of the first relation joins the same concept as each arc of the second relation.

A conceptual graph that represents a possible situation in the external world and is meaningful is termed a c*anonical graph*. Sowa defines four rules for canonical graph formation. These four rules are as follows: copy, restrict, join, and simplify.

Copy refers to making an exact copy of a conceptual graph. Since an exact copy of the entire graph is required, this implies that the components of the graphs, concepts and relations, must also be able to be copied. Restrict implies that for any concept in a graph, the type of the concept may be changed to its subtype, and a generic marker may be changed to an individual marker, provided the referent of the concept conforms to the same type before and after the changes.

The join rule says that, in two graphs, if a concept in the first graph is identical to a concept in the second, then one of the identical concepts may be deleted, and all the arcs of the deleted concepts may be linked to the other identical concept. The simplify rule states that, in a graph, if two conceptual relations are duplicates, one of them, together with all its arcs, may be removed from the graph.

A *schema* of a concept is described as "the basic structure for representing background knowledge." Schemata are said to incorporate domain-specific knowledge about entities, attributes and events in the real world. Schema graphs provide information about the concept in terms of the conceptual relations that the concept may be incident to, and the concepts that are linked to these conceptual relations. For example, the schema for a bus (used in computer and digital logic systems) is shown in Figure 2-1.

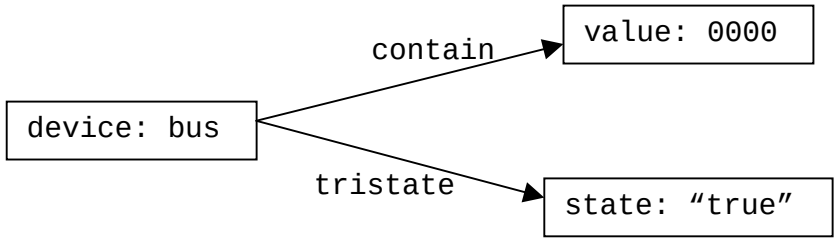

Figure 2-1–Schema for a "bus" concept.

Figure 2-1 shows that the bus may be described by the value it contains, and whether the bus is tristated or not. What Figure 2-1 effectively does is to describe the "bus" concept in terms of its associated conceptual relations, and the concepts linked to those relations.

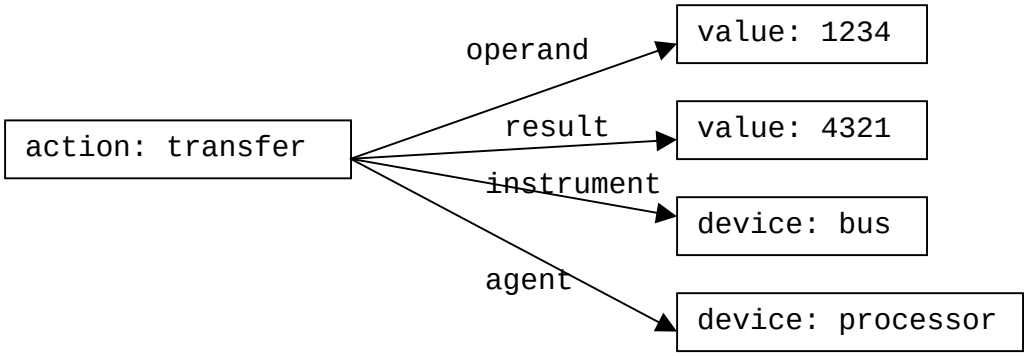

Figure 2-2–A graph that uses the “bus” concept.

Figure 2-2 shows a graph that uses the bus as a concept. When a conceptual graph contains a concept for the bus, the concept is expanded by its schema graph, by performing a join operation. Figure 2-3 shows the graph obtained by joining the graphs of Figures 2-1 and 2-2.

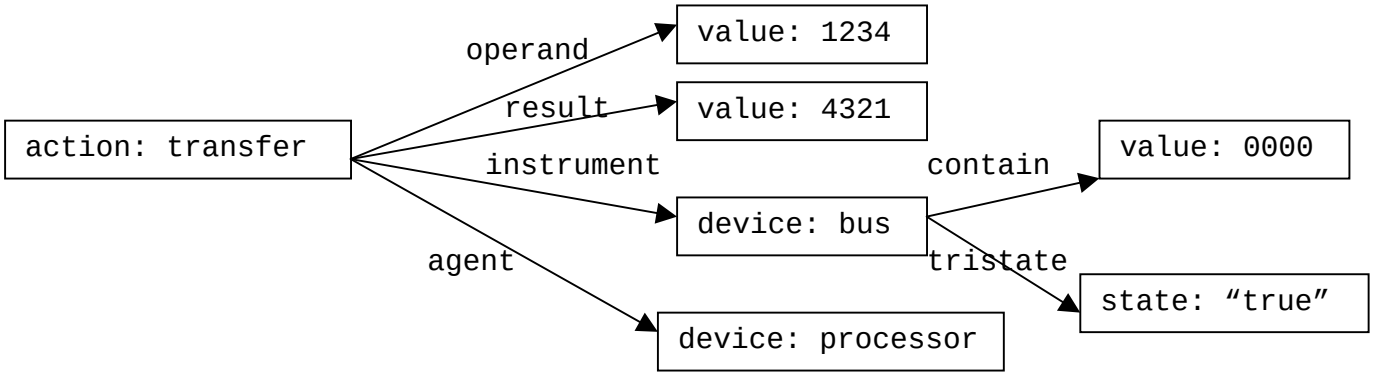

Figure 2-3–Graph of Figure 2-2 after performing the join.

As described above, when two graphs have an identical concept, in this case the “bus” concept, the join operation removes one of the two identical concepts and joins all the arcs of the removed concept to the other identical concept.

2.2 Conceptual Modeling and Simulation

In his book, Sowa says that in order to serve as a basis for thinking, conceptual graphs must be used in computation. The use of *schema* and *actors* for executing conceptual graphs are discussed in [Sowa, 1984]. A *schema* is described as "the basic structure for representing background knowledge." Schemata are said to incorporate domain-specific knowledge about entities, attributes and events in the real world. An *actor* is defined as a process capable of responding to messages by performing some service, and then generating messages for other actors.

These actors form "finite, connected, directed, bipartite" data-flow graphs with concept nodes of conceptual graphs [Sowa, 1984]. Relations between actors and concepts are determined with respect to the actors, hence all relations leading into actors are called *input arcs*, and those leading out of actors are called *output arcs.* This method of executing conceptual graphs uses computationally efficient data-flow graphs. This method also requires the addition of actor nodes to conceptual graphs. In addition, the creation of data-flow models from natural language is itself a difficult task.

Delugach considered state-transition models for execution of conceptual graphs [Delugach, 1990; Delugach, 1991]. He argued that as the conceptual graph makes a transition from one state to another, the old state had to be destroyed, and a new state created. He suggested the addition of a *demon* node to combat this problem. [Raban and Delugach, 1997] extended this approach with the introduction of assertion type events. The algorithm adopted here avoids this problem by making states either true or false.

Lukose [Lukose, 1993] suggested that graphs be extended in an object-oriented manner, with actor nodes being associated with executable procedures. These procedures could be invoked by messages. The execution of the procedures would result in a state that could serve as a condition for other actors. The execution is controlled using problem maps. Though his approach associates an action with an executable procedure, it does not specify how event and proposition types may be handled. The present approach suggests

explanations for the handling of event and state concepts. Also, the control of execution in the present algorithm is performed using conceptual relations and enabling conditions.

Simulation of conceptual graphs has been considered more generally [Bos et al., 1997]. These authors have considered concepts like actions, states and events in a state transition system. States and events are used as enabling conditions for action concepts. Actions are either recursively defined, or may be joined by temporal relations. The authors defined events as changes in states. Transition concepts were used to link these state changes, invoked by actions by using *effect* relations. Simulation proceeded in steps, where a step identified the set of enabled actions, and selected one for execution. Simulation time advanced (in fixed increments) with each step. When multiple actions are enabled in a step, the user resolved which action is to be executed. Simulation also detected unreachable actions and inconsistencies. The present approach is similar, the principal difference being the execution of all enabled actions in a step. Instead of transition concepts, this algorithm represents states as nested graphs that have to be evaluated for their veracity.

2.3 Discrete Event Simulation

We now look at some of the commonly used terminology with respect to discrete systems and discrete event simulation. The process being modeled is referred to as the *system*. The *system* is defined to be "a collection of entities, e.g. people or machines, that act and interact together toward the accomplishment of some logical end [Schmidt and Taylor, 1970]." The description of the system takes the form of logical relationships between the various components of the system. Such relationships constitute a *model* of the system [Law and Kelton, 1991]. The *state* of the system is defined as "that collection of variables necessary to describe the system at a particular time, relative to the objectives of a study [Law and Kelton, 1991]." The behavior of the system at any point in time depends on its state at that point in time. A *discrete* system is one in which the state variables of the system change instantaneously at separated points in time.

In a simple logic simulator, behaviors of circuit components are usually stored in a library. High level simulators allow users to write processes that characterize the behavior of the system [Lipsett et al., 1989]. When the system is started, all processes execute once. After startup, processes execute only when stimulated. Stimulation of processed take place by input signals, which are provided by the user in the form of a test bench. Simulation takes place in cycles, and in each cycle, all stimulated processes execute and generate output signals. A general discussion of discrete event simulation is covered in [Schriber and Brunner, 1996].

A general modeling notation called state-charts was proposed in [Harel and Naamad, 1995]. This notation has a clearly defined procedure for simulation. State-charts are an extension of finite state machines. The states in a state-chart may either be hierarchical or parallel. Since a state chart may be described using parallel states, the total system may be represented as being multiple sub-states, one in each of the parallel states, at a given time. State transitions may be triggered and/or preconditioned. These transitions occur only when the enabling condition is true and the triggering event occurs. Actions may be initiated by a state transition. Alternatively, actions that are associated with the system residing in a given state are performed when the system is in that state. Executing actions have the ability to generate new events.

2.4 A Prototype Conceptual Graph Simulator

We have looked both at traditional techniques for simulating conceptual graphs and at discrete-event simulation techniques. We now look at some work that proposes how discrete event simulation techniques may be adapted to conceptual graph simulation.

Cyre has addressed the issue of “directly executing conceptual graphs,” and the mechanics of performing such a simulation [Cyre, 1998]. He has presented a model for executing general conceptual graphs without the addition of special actors, unlike [Sowa,

1984]. The creation and destruction of states proposed by Delugach [Delugach, 1991; Delugach, 1990] is avoided in this algorithm by making states either true or false.

The algorithm proposed by Cyre [Cyre, 1998] was inspired by the state-charts approach. Cyre suggested an algorithm for the simulation process by describing a series of steps that a simulator needs to perform. The execution model he described considered the interaction between behavior concepts (actions, events, states) and values. The way a simulator handles such concepts have been explained in detail. Detailed descriptions of the simulation semantics for each type of concept were provided. The conceptual graph simulation algorithm described by Cyre was event driven. He also coined a term *incident* to describe the basic entity that the simulation algorithm acts upon [Cyre, 1998]. Incidents propagate through the graph by means of conceptual relations. Cyre analyzed the signatures of frequently used conceptual relations and gave guidelines on how such relations affect the generation of new incidents.

A prototype of the conceptual graph simulator was implemented using Prolog. This prototype was an event-driven simulator, designed for simplicity. The Prolog simulator is an implementation of the algorithm by Cyre [Cyre, 1998]. It uses Prolog files for the simulator and the conceptual graphs.

This chapter has been devoted to the survey of relevant previous work. We next discuss the general design of the simulator, and look at the issues involved in adopting an event-driven approach to conceptual graph simulation.

Chapter 3 Design of the Simulator

This chapter covers the issues involved in the design of the simulator. We first examine an example conceptual graph, and then explain the associated terms and their meanings. The role of each type of concept in simulation is examined, and the means of making each concept type simulatable are considered. We then look at how the simulator performs its functions using queues. The order of processing of incidents is determined, and the steps to be taken for processing each type of incident are discussed. Finally, the role of conceptual relations in conceptual graph simulation is considered.

3.1 Terminology

A conceptual graph (CG) consists of a fixed set of concepts, and a set of conceptual relations between these concepts. A good analogy for such a graph would be a road atlas, where the concepts correspond to the cities and the conceptual relations to the highways connecting them. Conceptual graphs are used for representing mechanisms, and their responses to stimuli. Typically, the concepts represent the following: the devices in the mechanism, the set of actions that these devices can perform, the set of states that these devices can have, and the set of values that these devices can store. An example conceptual graph is given in Figure 3-1. Section 2.1 discussed conceptual graphs and their properties in greater detail.

The CG in Figure 3-1 represents the following English sentences: “Keep incrementing a counter every 10 seconds while the count is less than 6. When the count reaches 6, signal the alarm.” A typical use of the above graph would be in a model of a fire-alarm detector, which detects smoke signals every 10 seconds and sets off an alarm if it consistently detects smoke signals for 60 seconds. We shall use the CG in Figure 3-1 to approach the design for the conceptual graph simulator.

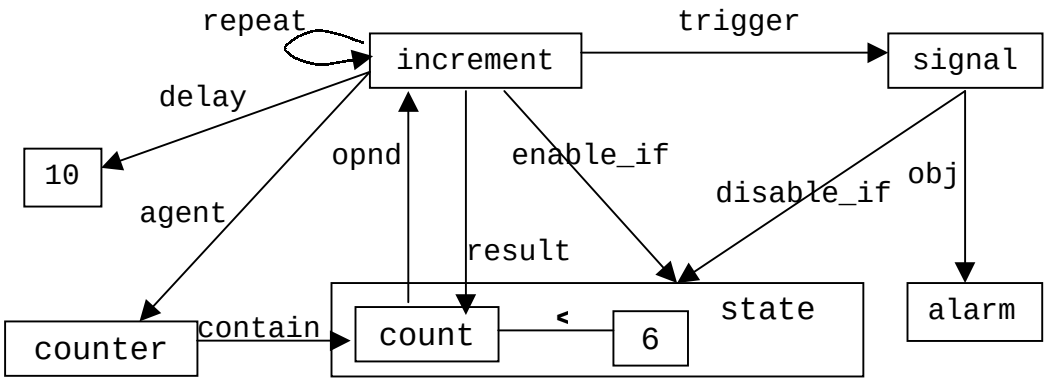

Figure 3-1–An example conceptual graph for high-level design purposes.

The concepts in the conceptual graph of Figure 3-1 are *counter, increment, state, count, '6', '10', signal,* and *alarm*. The relations in the above CG are *agent, opnd, contain, result, enable_if, disable_if, delay, repeat, <, trigger* and *obj*.

The *count, '6'* and *'10'* concepts in Figure 3-1 are termed *value concepts*. By saying that a particular concept is a value concept, we imply that the concept has a value associated with it. For example, the *'6'* concept has an attribute six (6) associated with it. The *count* concept also has a value associated with it. This value holds the current value of the *count* concept. In summary, each value concept represents a value.

The *counter* and *alarm* concepts of Figure 3-1 represent physical devices; such concepts are referred to as *device concepts*. Device concepts are used to provide a physical description of a mechanism. These concepts are useful for representing the mechanism in terms of its constituent device components. Such concepts may have an *operational mode*; the counter may either be reset or enabled, the alarm may either be ringing or off. Each device concept has its own set of operational modes. The operational modes of one device may not be applicable to another device; the counter is ringing, or the alarm is reset do not make sense.

The concept that is labeled *increment* in Figure 3-1 is termed an *action concept*. An action concept is one that performs a certain action. In Figure 3-1, the *increment*

concept performs the action of incrementing the *count* concept. All action concepts have an *activity*; that is, the action is either executing (active), not executing (inactive) or suspended. Action concepts are used to perform a number of functions: these include decoding instructions, incrementing/decrementing values, reading/writing values from value concepts, and fetching/storing operands. As discussed in section 1.1, invoking an action procedure associated with the action word performs these functions.

The *signal* concept of Figure 3-1 is an *event concept*. The main difference between action and event concepts is that we assume an action concept, such as incrementing a counter, may take a finite amount of simulation time to execute, whereas an event concept, such as signaling an alarm, executes instantaneously.

The concept labeled *state* in Figure 3-1 has veracity. What we mean by this is, the *state* concept has a status of either true or false. Such concepts, that represent a status, are termed *state concepts*. The status of a state concept can either be a veracity state (like true or false), or it can represent the activity of an action (like active, inactive, or suspended). Alternatively, the status may hold the mode of a device (like busy, ringing, reset, etc.).

In summary, value concepts have a value associated with them, state concepts have associated states, device concepts have operational modes, and action concepts have associated activities. Such associated values and states, which are changeable, are called *attributes*. An *event* is a change in an attribute of a concept. Such changes are caused by event and action concepts when they execute. These changes propagate along the conceptual relations in the graph, and their effects are felt by other concepts. Event concepts execute instantaneously, whereas action concepts have duration. Actions execute by invoking their corresponding action procedures.

Simulation of the above graph can be implemented in various ways, as seen in Chapter 2. We base our simulator design on an event-driven approach. This approach has been used in various commercial and academic simulators. [Harel and Namaad, 1995] used the event-driven approach in the design of their StateMate© system. [Lipsett et al.,

1989] also employ the even-driven simulation method in their design of a Hardware Description Language (HDL) simulator.

The term *event* that appears in the event-driven approach is distinct from the *event* in the event concept mentioned before. In order to distinguish these events (attribute changes) from event concepts, we define a term *incident*. An *incident* is the element processed by the simulation algorithm. An incident refers to a change (either in value or state) in an attribute of a concept, and each incident is processed at a definite time. The term *incident* is treated in more detail in [Cyre, 1998].

3.2 Concept Types and Their Simulatability

Concept types may be classified as being simulatable or non-simulatable. Simulatable concept types are those concept types that play a direct role in simulation. What do we mean by saying that some concepts play a direct role in simulation, whereas others do not? We need to examine this more closely.

Let us consider value concepts and state concepts. These types of concepts have attributes that may undergo change during the simulation. The mutable attributes of these concept types can affect the way in which the simulation runs. Such concept types play an important role in simulation. They have a direct effect on the simulation, and the simulation has a direct effect on the attributes of these concept types. Hence, such concept types are simulatable.

Event concepts and action concepts have certain behaviors ascribed to them. Such concept types are executable. Executing these concept types leads to changes to the attributes of the value and state concepts. Also, the initiation and/or inhibition of other event and action concepts are controlled by the execution of event and action concept types. Therefore, event and action concept types are also simulatable.

Device concepts, on the other hand, do not play a direct role in simulation. They serve three primary functions in conceptual graphs. Firstly, they are responsible for the physical containment of value concepts. Secondly, the operational modes of these device concepts may serve as state concepts (which in turn may serve as enablements or disablements for event and action concepts.) Thirdly, they serve as the agents responsible for performing the execution of event and action concepts. Hence, device concepts are not simulatable, but the attributes of these concepts (such as the values they contain, or their operational mode) are simulatable.

As described above, value, state, event and action concept types are simulatable. Their purpose in the conceptual graph is to provide a logical description of the system being modeled. They present a functional view of the system, and depict the behavior of the system in response to external stimuli. Device concepts strive to provide a clearer picture of the physical description of the system, as opposed to the logical description provided by the other concept types. With respect to incidents that were discussed in section 3.1, we may now draw the conclusion that only the simulatable concepts of the CG may have incidents. Non-simulatable concepts do not directly affect simulation, and hence do not have incidents. Incidents are thus of four types: value incidents, state incidents, event incidents and action incidents.

We next consider each concept type from the simulation point of view. We decide if the concept type can be simulated as is, or whether the conceptual graph needs to be augmented with more detail in order to simulate the concept type. The issues of adding attribute concepts and relations to the conceptual graph in order to make it simulatable, referred to as *preparing* the graph, are addressed. Section 3.6 considers the creation of action procedures based on schema graphs for action concepts. Section 3.7 discusses the expansion of device concepts using schema graphs.

3.2.1 Value Concept Type

The value concept type is simulatable. Its function in the conceptual graph is to hold values. These values are mutable, that is, subject to change during the course of simulation. To be able to simulate the graph, we need to ensure that each value concept has a value associated with it. This value needs to be stored as part of the value concept.

When we examine [Sowa, 1999], we see that there are two available places for storing a value: the first is the *referent field* of the concept, and the second is the *concept comment*. The *referent* of the concept is the entity or entities that a concept refers to. The *referent field* of a concept is the area in a concept where the referent is specified. The *concept comment* is an area of the concept that holds a comment about the concept. These places are shown in Figure 3-2.

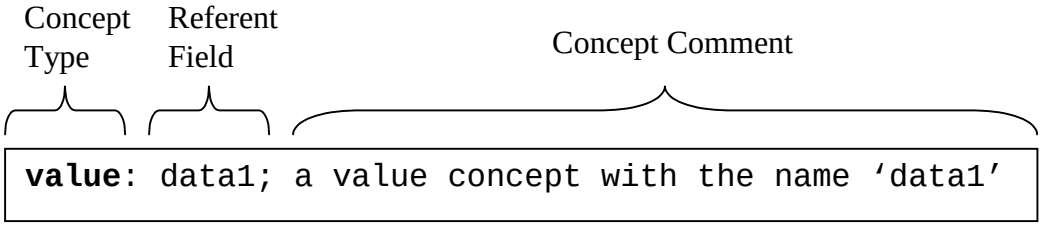

Figure 3-2–A value concept that does not store a value as yet.

Since the referent field is used to refer to entities that identify or describe the concept, it cannot be used to store values. It may be possible to use this field in some concepts, which do not refer to other entities, but we need to maintain uniformity for all concepts, hence the referent field is not used. The concept comment is an excellent choice for storing values. This way, these stored values will not affect the main graph–they are just a part of the comment–but they can be used in simulation. However, device concepts have a number of mutable attributes, all of which cannot be stored in the concept comment. Therefore, it was decided not to use the concept comment for storing attributes. As no part of the concept could be used for storing a value, it was decided that a new concept would be added to the conceptual graph, and related to the value concept by using a unique relation.

For each value concept present in the graph, another new concept is introduced into the graph. The sole purpose of the new concept is to hold a value for the value concept. Apart from the value concept that it holds the value for, this new concept does not interact with or relate to any other concept in the graph. The value concept and the new concept are related by means of an *eval* relation, meaning that the value concept "evaluates to" the value stored in the new concept. Figure 3-3 shows how the new concept and relation are added, thus associating a value with the value concept.

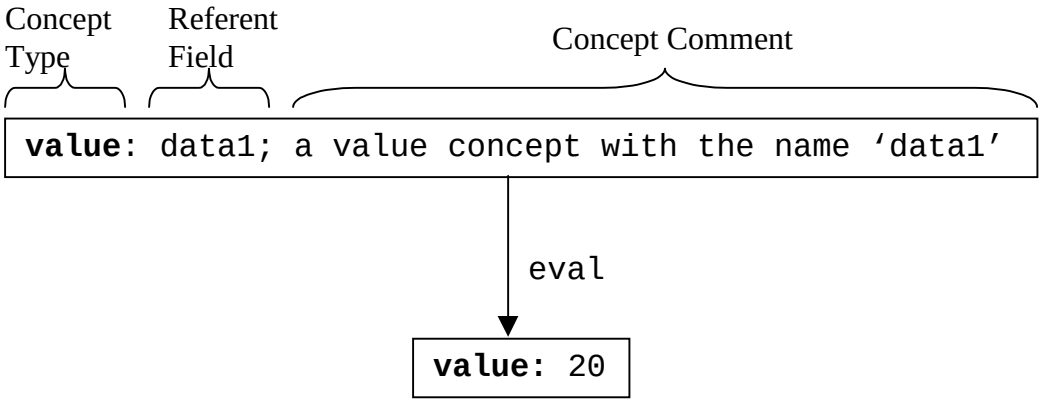

Figure 3-3–A value concept and it's associated value.

3.2.2 State Concept Type

The state concept type is also simulatable. The state concept is used to hold the state of the system. These states of the state concept are also subject to change during the course of simulation. Typical examples of states stored in a state concept are as follows:

a) the *operational modes* of devices, such as "ready", "busy", "reset", or "disabled"
b) the *activity* of actions, such as "active", "inactive", or "suspended"
c) veracity values like "true" or "false"

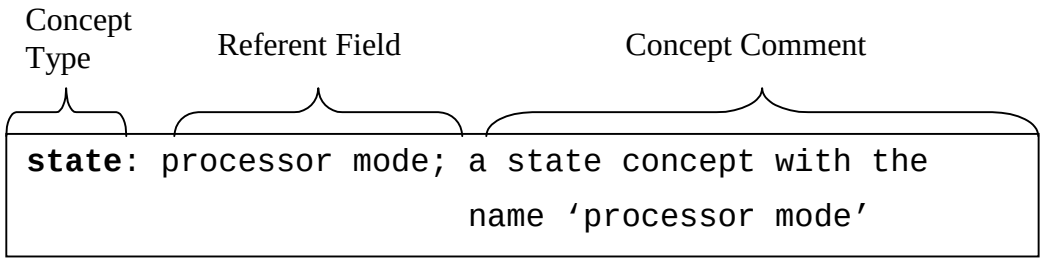

Figure 3-4–A state concept that does not store a state as yet.

Figure 3-4 shows a state concept that has not yet been associated with a state. As with the value concept, it was decided that no part of the state concept might be used to store the state. So, all the state concepts in the graph had to be prepared for simulation. For preparing each state concept, a new concept had to be introduced in the graph, and the new concept had to be related to the state concept using the *eval* relation. Figure 3-5 shows the state concept after it is associated with a state using the *eval* relation.

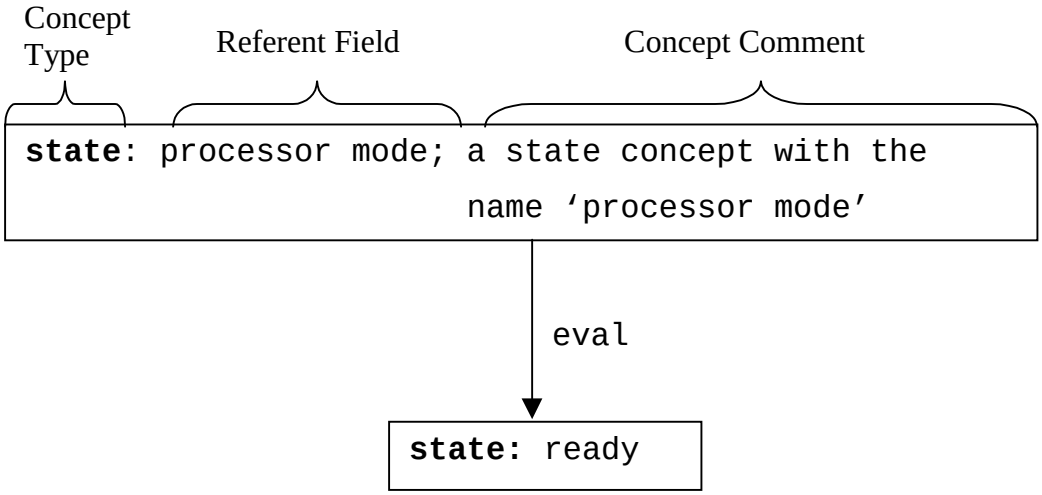

Figure 3-5–A state concept and it's associated state.

3.2.3 Event Concept Type

The event concept type does not store any values or states. Also, the execution of an event concept takes place instantaneously. The instantaneous execution of the event concept is referred to as the *firing* of the incident. Because of these two properties of the event concept type, no preparation is needed in order to simulate the event concept type.

3.2.4 Action Concept Type

Action concepts take a finite amount of time to execute. We define a term called the *activity* of an action. The *activity* of an action says whether the action is executing, not executing, or suspended, at a given simulation time. To represent the activity of an action, each action concept needs to be associated with a state concept. This state concept stores information on whether the action is currently executing, inactive, or suspended. Figure 3-6 shows an action concept that has not yet been associated with its activity.

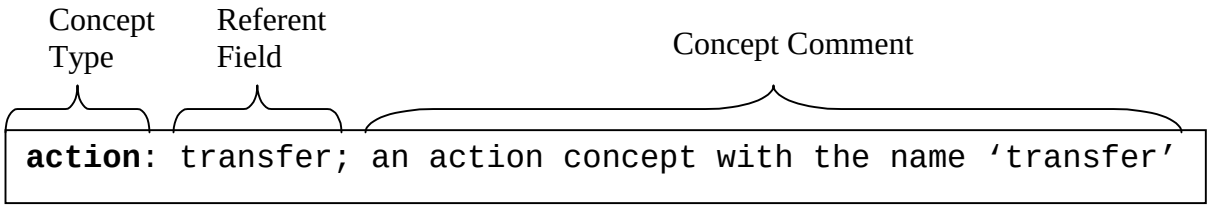

Figure 3-6–An action concept not yet associated with its activity.

As with value and state concepts, no part of the existing action concept may be used to store its activity. A new state concept is added for each action concept in the graph. This new state concept is related to the action concept by means of a unique relation, designated as the *mode* relation, meaning that the action concept has an activity indicated by the newly added state concept. Figure 3-7 shows an action concept that is associated with its activity by the *mode* relation.

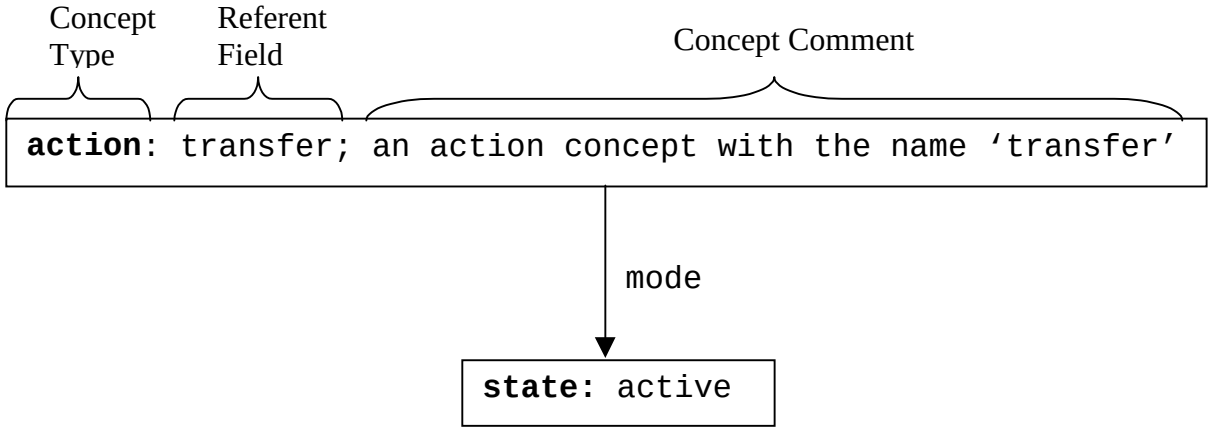

Figure 3-7–An action concept and it's associated activity.

3.2.5 Device Concept Type

The non-simulatable device concept type is used to describe physical devices, such as processors, memory, registers and so on. Concepts of this type tend to play a passive role in simulation. As discussed earlier in this chapter, the three primary roles played by device concepts in a conceptual graph are to store values, affect the enablement or disablement of events and actions by virtue of their operational modes, and act as agents for executing actions. There are no incidents of the device concept type, such incidents are neither generated nor processed.

However, concepts of this type may have mutable attributes. Examples of such attributes include the operational mode of a processor, the value stored in a register, the location of a program in memory, and so on. These mutable attributes tend to play a vital role in the simulation, as these are entities that undergo a change of state or value. Device concepts by themselves are a passive part of conceptual graph simulation, but their related concepts play an active role in the simulation. The expansion of device concepts using schema is covered in section 3.7 of this document.

3.3 Simulation Using Queues

As discussed earlier in section 3.1, an incident refers to a change in an attribute of a concept. This change can be either a value or a state change. In addition, incidents can fire event concepts and affect the execution of action concepts. Each incident has a definite time at which it is processed. Incidents are generated by event and action concepts when they execute.

To keep track of all the incidents present, the simulator needs to maintain queues. The incidents present in the simulator may be classified into three categories. These three categories are as follows: current incidents (incidents to be processed at the current time), later incidents (incidents to be processed at a later time) and new incidents (that are generated due to the processing of current incidents). Therefore, the simulator needs to maintain three queues for handling incidents. These three queues are the current queue, the incident queue, and the future queue. The current queue is a queue of incidents that needs to be processed in the current simulation cycle. The incident queue holds incidents to be processed at later times. The future queue holds incidents that are generated when the current incidents are processed.

The simulator also needs to maintain the current simulation time. The current simulation time is used to ensure that incidents are processed in time-ascending order. The simulation time is measured with respect to the beginning of the simulation, which

occurs at time zero. Simulation time is monotonically increasing and cannot be reversed. The simulation time is increased after all the current incidents have been processed. It is increased by setting it to the processing time of the earliest executable incident in the incident queue. The act of the simulator processing all current incidents (at the current simulation time) is called a *cycle*. Often, incidents generated at the current simulation time may need to be processed at the current time itself (without delay). When such incidents exist, the simulator processes the newly generated incidents without advancing the simulation time. That is, a cycle does not necessarily advance the simulation time. The simulator may perform many cycles at a given simulation time. Simulation proceeds in the three-phase manner, as indicated below.

The simulation is initiated with a set of initial incidents, which have to be supplied externally (by the user). These incidents are put into the incident queue. The simulation time is then initialized to 0. The simulator then does the following steps:

a) it moves incidents that have to be processed at the current simulation time from the incident queue to the current queue,
b) it processes the incidents in the current queue, adding any newly generated incidents to the future queue, and when all the current incidents have been processed,
c) it moves the incidents from the future queue to the incident queue, ensuring that the incidents in the incident queue are in time-ascending order.

These steps are repeated until the end of simulation is reached. The end of simulation is reached when both the current and incident queues do not contain any incidents to be processed. Alternatively, simulation may be ended after a predetermined amount of time has elapsed since the beginning of simulation.

3.4 Processing of Incidents

We first discuss the order in which incidents have to be processed, and why such an order is needed. Once this order of processing has been decided, we look at the

processing of each type of incident in detail. When dealing with incidents, we shall refer to the concept that generates an incident as the causing concept, and the concept for which the incident is generated as the caused concept.

3.4.1 Order of Processing of Incidents

The current queue contains all the incidents that need to be processed at a given simulation time. We now decide the order of processing these incidents. We first need to determine whether these incidents have to be processed in any particular order or not. If a definite order is needed, we decide the order in which these incidents are processed.

The state of state concepts can be dependent on values, and hence changes in values need to be processed before changes in state can be processed. This precedence of value incidents over state incidents ensures that states are evaluated correctly. This is better illustrated in Figure 3-1. A change in the value of the c*ount* concept will change the state of the *state* concept, once the count reaches 6. Hence the value incident, representing a change in *count*, needs to be processed earlier.

Event concepts are highly sensitive to changes in values and states because states and values enable the events. If we processed event incidents before we processed state or value incidents, it may result in processing an event that should not have been processed, or vice versa. Hence, event incidents need to be processed after value and state incidents have been processed.

Events are the cause of starting, suspending, resuming, or terminating actions. So, the execution of action concepts is definitely dependent on the processing of event incidents. Processing action incidents before event incidents are processed may lead to some actions not being executed. Similarly, if an event concept is supposed to prevent an action concept from executing, by not processing the event incident before, we may execute an action that had to be suspended. Therefore, action incidents need to be processed only after all event incidents have been processed.

In summary, incidents have to be processed in the following order: value incidents, state incidents, event incidents and action incidents. It is important to remember that the above order of processing incidents holds good only when multiple types of incidents have to be processed at a given simulation time. The above order of processing events does not extend over different simulation times. We next look at the processing of each type of incident in detail.

3.4.2 Value Incidents

As noted earlier, value incidents need to be processed first as they may affect the state of state concepts. Value incidents cause a change in value. During the course of simulation, the simulator time reaches the time at which a value incident has to be processed. When this happens, the value incident is moved into the current queue and is processed. When a value incident is processed, the following steps are performed:

1) the value concept that needs to be changed is located in the graph
2) the value stored in the concept is changed

3.4.3 State incidents

State incidents need to be processed after all value incidents have been processed and before the processing of event incidents. State incidents cause a change in state. When the current simulation time matches the processing time of a state incident, the state incident is moved from the incident queue to the current queue, and processed. As in the case of a value incident, state incidents are also processed in two steps:

1) the state concept that needs to be changed is located in the graph
2) the state stored in the concept is changed

We can see that the simulator handles value and state concept types very similarly. This is because they essentially perform the same function of storing some information, which may be modified during the course of simulation, without time necessarily passing.

3.4.4 Event Incidents

As stated earlier, because of the instantaneous execution of the event concept, the execution of an event concept is referred to as *firing* the event. If an event concept is related to an action concept, firing the concept can initiate, suspend, resume or terminate that action. When the activity of an action changes, this change in activity is also treated as an event incident. Hence the initiation and termination of actions are also treated as events. This is further explained in section 3.5.2.

There is a fixed set of steps that needs to be followed when an event incident is processed. The steps to be followed are as follows:

1) the event concept that has to be fired is first located
2) if an enabling condition exists, and the condition is false, nothing further is done
3) the event concept is added to a list of fired events (reason explained below)
4) all outgoing propagational relations are processed, and incidents are generated

One significant point of interest is step 3 above. In this step, we add the firing event concept to the list of fired events. The reason behind this is also the transient nature of the firing of the event. The need for this is better illustrated with an example graph. Figure 3-8 shows a conceptual graph that necessitates the usage of the fired event queue. Suppose action A1 initiates action A2 only if both events E1 and E2 have fired. Since event incidents are executed before action incidents, the incidents for E1 and E2 will have fired before the incident for A1 is processed. Hence the event incidents no longer exist in the current queue. Action A1 has no way of knowing whether the events have fired or not, and will not initiate action A2. But this is wrong. Action A2 should be initiated because events E1 and E2 have already fired.

To keep track of the highly transient event concepts, we use a list of fired events. All events that fire go into the list of fired events. So, event E1 is added to the list of fired events when it fires. Similarly, event E2 is also added when it fires. When action A1 needs to decide whether the events have fired or not, it searches the list of fired events for

the presence of E1 and E2. It finds both the events in the fired event list, and therefore initiates action A2. Events fired at earlier simulation times should not affect the generation of incidents at the current simulation time. In order to ensure this, the list of fired events is cleared whenever the simulation time advances. In the case of a timeless model (when the model has no delays) this list of fired events is cleared when an action completes execution.

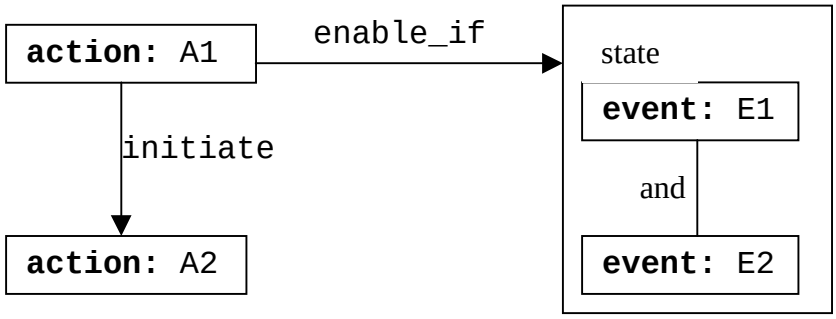

Figure 3-8–A graph that needs the fired event queue to simulate properly.

3.4.5 Action Incidents

By virtue of being executable and having duration (taking a finite amount of time to execute), action concepts are the most interesting concepts to deal with. Action concepts are responsible for performing all the actions that require a certain amount of time to be executed. Execution of action concepts changes values and states, triggers events, and controls the execution of action concepts. As discussed in section 3.1, action concepts are executed by invoking relevant action procedures. The name of the action procedure to be invoked is determined by the name of the action concept.

An executing action has duration. An action may be brought into execution by initiating it, an executing action may be suspended or terminated, and suspended actions may resume execution. Therefore, actions can be initiated, suspended, resumed or terminated. Depending on the conceptual relation that generates the action incident, there are four different types of action incidents. They are the “initiate action,” “suspend action,” “resume action,” and “terminate action” incidents. Each type of action incident affects the activity of the action in a different manner. When an action incident is

processed, these four types of action incidents must be taken into consideration. Therefore, for each type of action incident, a different set of steps needs to be performed.

We first consider the "initiate action" incident. What this incident implies is that the action needs to be put into execution. For this, the steps to be performed are as below.

1) the action concept that has to be initiated is first located
2) the name of the action concept is used to identify the action procedure to be called
3) the actual action is performed by calling the action procedure
4) all outgoing propagational relations are processed, and incidents are generated

The processing of an "initiate action" incident is performed in four steps. First, the action concept that the incident refers to is located in the graph, as seen in step 1 above. Each action has an action procedure, and this procedure has to be called to execute the action. The action procedure is found from the referent of the action concept, as seen in step 2 above. The functions performed by the action procedure are discussed in section 3.6 of this document. These functions include the generation of state incidents for the activity of the action. Step 3 represents the actual task that an action has to perform, by calling the action procedure. Finally, step 4 generates new incidents depending on the presence of outgoing relations.

We define here two terms. The *action initiation state incident* is the state incident that changes the activity of the action from "inactive" to "active," marking the beginning of the action's execution. The *action termination state incident* is the state incident that changes the activity of the action from "active" back to "inactive," marking the completion of the action's execution.

We next examine the "terminate action" incident. A "terminate action" incident implies that a currently executing action needs to be terminated at the current simulation time, without allowing it to reach its scheduled completion time. How is this achieved? First and foremost, the activity of the action needs to be changed from "active" to "inactive." But doing just this is not sufficient. We also need to remove all incidents for

which this concept was the causing concept. How do we know that these incidents exist? For an action, which has duration, the action termination state incident will be processed when the action completes its execution. Also, value incidents that are generated, by the execution of the action, will be processed when the action completes its execution. And the "terminate action" incident is being processed earlier than the scheduled completion time of the action. Therefore, all these incidents must be present in the incident queue. The following steps are performed to process a "terminate action" incident.

1) the action concept that needs to be terminated is identified in the graph
2) all incidents in the incident queue, future queue, and current queue which have the concept in step 1 as their causing concept are removed from those queues
3) a new state incident is generated to change the activity of the action from "active" to "inactive," scheduled to be processed at the current simulation time

The next type of action incident we look at is the "suspend action" incident. A "suspend action" incident implies that a currently executing action be suspended. The activity of the currently executing action has to be changed from "active" to "suspended." Suspending an action means that no value or state changes caused by this action after the current simulation time should take place. It also means, if any events and actions caused by this concept are to be processed at a later time, such incidents should not be processed.

In terms of the simulator, this translates to ensuring that no incidents for which this action is the causing concept are processed. However, these incidents may be required at a later time, if the suspended action resumes execution. Hence we need to store all the above incidents (value, state, event, and action incidents) in another queue. This queue is called the *suspended queue*. This queue stores all the incidents of actions that are currently suspended. The "suspend action" incident is handled as follows:

1) the action concept to be suspended in located in the graph
2) all incidents in the current, incident and future queues that have the concept in step 1 as their causing concept are moved to the suspended queue
3) a new state incident is generated, which sets the activity of the action to "suspended" from "active," scheduled to be processed at the current simulation time

Finally, the “resume action” incident needs to be examined. The “resume action” incident means that a currently suspended action needs to resume execution. This means the activity of the action needs to be changed from “suspended” back to “active.” Also, all incidents for which this action is the causing action must be retrieved from the suspended queue and inserted back into the incident queue. The steps for processing a “resume action” incident are as follows:

1) the action to be resumed is located in the graph
2) all incidents with this action as their causing concept are retrieved from the suspended queue, and inserted into the incident queue
3) a new state incident, for changing the activity of the action back to “active,” is generated, scheduled to be processed at the current simulator time

We define here two more terms. The *action suspension state incident* is the state incident that is responsible for changing the activity of the action from “active” to “suspended”, marking the suspension of the action’s execution. The *action resumption state incident* is the state incident that changes the activity of the action from “suspended” back to “active”, marking the resumption of the action’s execution.

The “suspend action” and “resume action” incidents present an interesting problem in terms of simulation time. This scenario is illustrated with the aid of the conceptual graph in Figure 3-9, which is depicted below. Figure 3-9 does not depict a simulatable graph; a lot of conceptual relations that are unnecessary to explain the simulation time problem are not represented. Only those concepts and relations relevant to the problem are shown.

The conceptual graph in Figure 3-9 represents the following English sentences “The transfer action, upon initiation, causes the sleep action. The sleep action, upon initiation, immediately suspends the transfer action. The sleep action, when it finishes execution, causes the wake action. The wake action, when it finishes execution, resumes

the transfer action. The delay associated with the transfer action is 15 time units, and the delay for the sleep and wake actions is 5 time units."

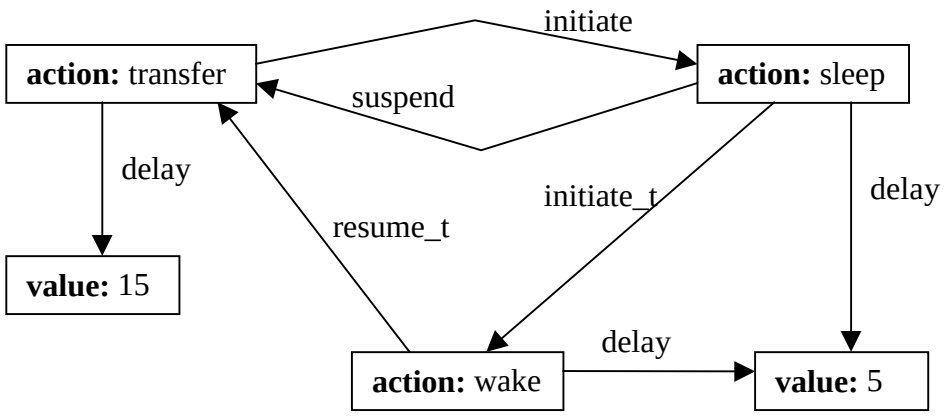

Figure 3-9–A conceptual graph with "suspend" and "resume" relations.

There is one point in this conceptual graph that is worth noting. The conceptual relations that end with "_t" are used to indicate that the relation comes into effect only when the action completes its execution. If the "_t" is not present, then it means that the relations come into effect as soon as the action begins execution.

A simulation of the above graph yields a clear picture on the simulation time problem mentioned above. At simulation time 0, let us assume that the transfer action is initiated. The transfer action has a time delay of 15 time units. The action initiation state incident is scheduled for simulation time 0, and the action termination state incident is scheduled at simulation time 15. Due to the presence of the *initiate* relation, the transfer action initiates the sleep action at time 0. Other incidents may be generated as part of the execution of the transfer action, but we neglect them for the present discussion, as they do not affect the current simulation time problem that is being discussed.

The sleep action has a *suspend* relation with the transfer action, so the sleep suspends the transfer action at simulation time 0.At this point, all incidents that were caused by the transfer action are moved to the suspended queue. In particular reference to this example, the action termination state incident is moved to the suspended queue. The

action suspension state incident changes the activity of the transfer to “suspended.” The sleep action also has an *initiate_t* relation with the wake action. The sleep action is associated with a time delay of 5 time units. Hence, the sleep action completes its execution at simulation time 5. At this time, the sleep action initiates the wake action.

The wake action has a *resume_t* relation with the transfer action, which implies that the transfer action resumes execution when the wake action completes its execution. The wake action is associated with a time delay of 5 time units. Since the wake action has been initiated at simulation time 5, it completes execution at simulation time 10. At this point in time, the transfer action resumes execution. The action resumption state incident changes the activity of the transfer to “active.” What we mean by this is all incidents from the suspended queue are now moved back to the incident queue. In particular, the action termination state incident is moved back to the incident queue.

The transfer action has been suspended for a total of 10 time units, as it was suspended at simulation time 0, and it was resumed at simulation time 10. The action termination state incident for the transfer action had earlier been scheduled for a simulation time of 15, so it is processed at simulation time 15, and marks the completion of the transfer action.

This means that the transfer action completed its execution at simulation time 15, even though it was suspended (was not executing) for 10 time units. Effectively, it means that the transfer action has executed in only 5 time units. But the conceptual graph says that the delay of the transfer action is 15 time units. There exists an inconsistency between what the graph says, and what the simulator has performed.

The inconsistency arises because of the fact that action termination state incident is still scheduled at the previous completion time, that is, simulation time 15. The fact that the action has been suspended for 10 time units has not been taken into consideration. Though the simulation time advanced by 10 time units, the suspended

incidents were still scheduled using the old simulation time. This change in simulation time needs to be reflected in the incidents of the suspended queue.

To ensure that the suspended events were correctly rescheduled when the action resumed its execution, a procedure for updating the processing time of all the suspended incidents was written. Whenever the simulation time changes, the change in simulation time is calculated. This time difference is then added to the processing time of each suspended incident.

With this procedure in place, when the graph shown in Figure 4-8 is simulated, the simulation continues as before until the transfer action resumes execution. This happens at simulation time 10. When this happens, due to the procedure mentioned above, all the suspended incidents now have a processing time that is 10 time units more than their previous processing time. In particular, the action termination state incident is now scheduled to be processed at simulation time 25 (an increase of 10 time units over the earlier time of 15). Therefore, this incident is processed at simulation time 25, and since the action has been suspended for 10 time units, the action begins at time 10 and completes at time 25, giving a time delay of 15 time units. This is consistent with the value specified in the graph.

With this, we end the sub-section describing the processing of action incidents. We have looked at the details of processing each type of action incident. We have also resolved a simulation time problem with respect to the *suspend* and *resume* relations. With this, the section on processing incidents also comes to an end. We have outlined the steps that need to be taken in order to process incidents of each type of concept.

3.5 The Role of Conceptual Relations

We now discuss the generation of new incidents. The processing of event incidents fires event concepts. The processing of action incidents affects the execution of action concepts. Firing events and executing actions process their conceptual relations.

These conceptual relations between concepts are used to indicate how the execution of one concept affects its neighboring concepts. Thus, conceptual relations provide a means for incidents to propagate through the graph. Figure 3-10 shows a conceptual graph that depicts the following English sentences “Using the bus, the memory keeps incrementing every 15 ns until it reaches a value of five. At this point, it stops incrementing and sends a signal which toggles the LED after 10 ns.”

Conceptual relations may be classified into two types, *executional* and *propagational*. Executional conceptual relations are relations that play a role in the execution of events and actions. These relations are used to query the value of value concepts, or the state of state concepts. Executional conceptual relations do not bring about changes in the graph; they are used for gathering information needed for the execution of events and actions. Propagational conceptual relations are those that concern the generation of new incidents. When firing events and executing actions processes such relations, the executing concepts generate new incidents.

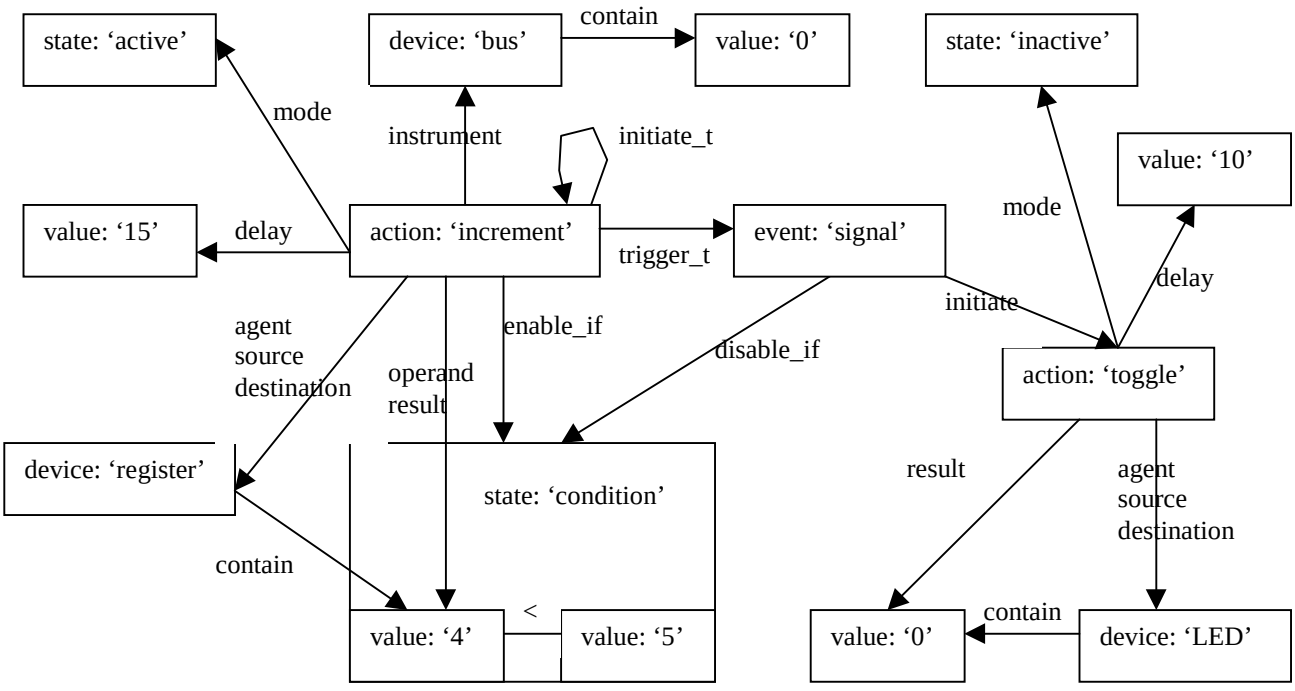

Figure 3-10–An example conceptual graph for explaining conceptual relations.

Examples of executional conceptual relations are *delay*, *enable_if*, and *operand* relations in Figure 3-10. Among these, the *delay* relation is responsible for getting the delay associated with the action. The *enable_if* relation is responsible for checking whether the executing concept is enabled or not. Similarly, the *operand* relation is responsible for gathering information about the operand for the action. The important point is, executional conceptual relations do not change the graph. They are responsible only for gathering information for the execution of the concept.

Examples of propagational conceptual relations are the *initiate*, *trigger_t*, and *result* relations of Figure 3-10. Each of these relations is responsible for generating new incidents. The *initiate* relation of Figure 3-10 is used to initiate the *toggle* action by generating an "initiate action" incident. The *trigger_t*, relation in the figure is used to generate a new event incident for the *signal* concept, and the *result* relation is used to generate value incidents for the *increment* and *toggle* actions.

We first examine executional relations in detail, and see how they affect the processing of incidents. Next, propagational relations are examined, and the steps for generating new incidents are discussed.

3.5.1 Executional Conceptual Relations

Executional conceptual relations gather enough information for the execution of event and action concepts. We first consider the execution of actions by the simulator. As part of the execution of these actions, we further classify relations into "essential" and "non-essential" relations.

As mentioned earlier, when an action concept is processed, the simulator calls an action procedure. The action procedure is specific to each action and tells the simulator precisely what must be done in order to simulate that action. This is best explained with the help of an example. Consider the example conceptual graph of Figure 3-10. The main concept is the action concept *increment*. The user supplies the simulator with a set of initial incidents to begin simulation. These initial incidents tell the simulator which

concept or concepts the simulation should begin with. In this case, an “initiate action” incident for the *increment* concept is supplied as an initial incident to the simulator. From this incident, the simulator picks up the action concept *increment.* To process the action incident, the simulator calls an action procedure for performing the increment action.

In the example graph in Figure 3-10, the essential executional relations for the *increment* action are the *operand* and *result* relations. They are essential, because they specify which concept is responsible for the increment action, and where the data for the increment action comes from. Without these two relations, it will be impossible to simulate the conceptual graph. The non-essential executional relations for the *increment* action are agent, *delay*, *source*, *destination*, *instrument* and *enable_if* relations. These relations are not essential for the simulation; the simulation can progress even in their absence. Based on a schema for an increment action, the action procedure for increment is designed to check for the following essential relations:

1) an *operand* relation to a value concept to get the value being incremented
2) a *result* relation to a value concept to store the value after it is incremented

The action procedure checks for all the essential relations and finds that all of them are present. So it is able to increment the register.

Consider the situation when the *toggle* concept needs to execute. As in the case of the increment action, the toggle action has an action procedure, which checks for all the essential relations for executing the toggle action. The toggle action needs the same set of essential relations as the increment action, *operand* and *result*. However, in this case, it finds that the toggle action does not have an *operand* relation. Hence the procedure is missing an essential relation, and hence cannot continue with the simulation. The simulation stops at this point, and the user is issued an error message, saying that the graph cannot be simulated any further, as it lacks the *operand* information. The user needs to now go back and edit the graph and include the *operand* relation. Only then will the simulator be able to simulate the above graph successfully.

Non-essential relations also have a very well defined role in which they affect simulation. The distinction between essential and non-essential relations does not lie in whether they affect the simulation or not. The distinction between the essential and non-essential relations depends upon whether or not simulation can be performed in the absence of these relations. The ways in which some non-essential relations affect simulation are discussed below.

A frequently occurring non-essential relation in these graphs is the *delay* relation. This relation links an action concept to a value concept that indicates the delay after which the effect of the action's execution is perceived in the graph. Consider the example of Figure 3-10, where the increment action has a delay of fifteen time units. The existence of the *delay* relation means that the increment action takes fifteen time units to execute. The *delay* relation affects the simulation by scheduling newly generated incidents (due to this action's execution) fifteen time units after the current time.

For a better understanding, let us consider a situation when the register contains a value of four, and the current simulation time is 't'. When the increment action executes, it generates a value incident, so that the value contained in the register is changed to five. This value incident is processed when the simulation time reaches a value of 't + 15'. The register contains the new value of five, fifteen time units after the incident is generated. The *delay* relation is very useful when modeling real world processes like memory reads/writes, which always need finite amounts of time to execute.

In the absence of a *delay* relation, the simulator understands that it is simulating a timeless model. A timeless model is one with no delay elements. When timeless models are simulated, the entire simulation is performed at time zero. The simulation proceeds in steps, each step corresponding to one simulation cycle. Actions that have zero delay get initiated in one cycle, and complete in the next cycle. An *empty cycle*, one that processes no incidents, separates the execution of one action from the execution of the next. As discussed earlier, when simulating a timeless model, the list of fired events has to be cleared in between the execution of two actions.

Another non-essential relation in Figure 3-10 is the *instrument* relation. The *instrument* relation says that the increment action takes place using the bus. Therefore, this relation is indicative of the instrument used to perform an action. This relation is not essential, since the increment action can be simulated even in the absence of an *instrument* relation. However, when this relation is present, it needs to be handled. The *instrument* relation is handled as follows. The increment action uses a device concept bus as its instrument. This device concept is associated with a value, using the *contain* relation. The *instrument* relation is simulated by ensuring that the value contained in the bus concept reflects the value being stored in the result concept (after incrementing). This reflects that the bus acts as the instrument in the incrementing of the register.

A third commonly occurring non-essential relation is the *enable_if* relation. This relation is used to link an action or event concept to a state, which serves as an enablement for the action or event concept. States may be represented as state concepts that have associated states. Alternatively, they may occur as nested (one within another) graphs that need to be evaluated. In Figure 3-10, the *state* concept is a nested graph. In such cases, the simulation of the outer conceptual graph requires the evaluation of the inner (nested) conceptual graph first.

We next discuss the propagational conceptual relations, and the methods of generating incidents when such relations are processed.

3.5.2 Propagational Conceptual Relations

The set of propagational conceptual relations covers those relations that are responsible for generating new incidents. The *trigger* relation deals with the generation of event concepts. There are four propagational relations that deal with action concepts. They are the *initiate*, *suspend*, *resume*, and *terminate* relations. Apart from these, the *result* relation is used to generate value and state incidents during action execution. We shall look at each of these relations and determine the steps to be taken when the relation

is encountered. We shall refer to the concept that generates an incident as the causing concept, and the concept for which the incident is generated as the caused concept.

During the course of simulation, when the value of a value concept needs to be changed, a new value incident has to be generated. The steps followed by the simulator for generating a new value incident are as flows:

1) in the graph, the value concept that needs to be changed is located
2) the concept responsible for generating this incident is also located
3) a new value incident for the concept found in step 1 is generated

Similar to the generation of a new value incident, the generation of a new state incident also takes three steps:

1) in the graph, the state concept that needs to be changed is located
2) the concept responsible for generating this incident is also located
3) a new state incident for the concept found in step 1 is generated

The *trigger* relation deals with the generation of event incidents. During the course of simulation, when a *trigger* relation is processed, an event incident is generated. This event incident holds the information as to which event is triggered, and the processing time of the incident. A new event incident is generated as follows:

1) the event concept that needs to be triggered is located in the graph
2) the concept responsible for generating this incident is also located
3) a new event incident for the concept found in step 1 is created, scheduled to be processed at the current simulation time

The executions of action concepts have duration, unlike event concepts. Since their execution is not instantaneous, there is a possibility that executing actions may be suspended before they complete execution, suspended actions may be rescheduled to resume execution, or executing actions may be terminated. Accordingly, we have four possible types of action incidents that affect action execution: incidents for initiating, suspending, resuming and terminating actions.

These action incidents may be generated either by event or action concepts. The conceptual relation between the causing concept and the caused concept determines whether the caused action is initiated, suspended, resumed, or terminated. Hence, this relation determines which type of incident, among the four types, should be generated. When a new action incident has to be generated, the following steps are taken.

1) the action concept for which the incident is being generated is identified
2) the causing concept is also identified in the graph
3) the relation between the causing and caused concepts determines the type of incident
4) a new action incident is generated for the concept identified in step 1, with the type determined in step 3, scheduled to be processed at the current simulation time

Table 3-1–Some conceptual relations and their consequent incidents.

Entity	**Incident Relation**	**Consequent Incident**
Event Concept	initiate	initiate action
	resume	resume action
	suspend	suspend action
	terminate	terminate action
	trigger	event
Action Concept	initiate	initiate action
	resume	resume action
	suspend	suspend action
	terminate	terminate action
	trigger	event

Hence, the presence of an *initiate* relation generates an “initiate action” incident. A *suspend* relation generates a “suspend action” incident. A “resume action” incident is generated due to the presence of a *resume* relation. Finally, a *terminate* relation is responsible for generating a “terminate action” incident. Table 3-1 shows some

propagational relations and the incidents generated due to these relations. The entity responsible for generating these incidents is also shown in the first column.

One important point to note in the above steps is that the generated incident is always scheduled at the current simulation time. When the causing concept is an event concept, the above steps work fine, as the event executes instantaneously. But suppose the causing concept is an action concept. Then, if we want to generate the incident when the action completes execution, or is suspended or resumed, how do the above steps provide for such situation? These situations are handled using *relation modifiers*.

In the graph being simulated, propagational relations may occur in their base form (as *initiate*, *suspend*, *resume*, *terminate*, or *trigger*). They may also be modified by the presence of a *relation modifier*. Relation modifiers for propagational relations take the form of an underbar "_" followed by a single letter. The letters that may follow the "_" are 's', 'r' and 't'. They respectively represent the suspension, resumption and termination of the action's execution. A modified propagational relation has a relation modifier appended to the end of the relation. Examples of modified propagational relations are suspend_t, resume_r and trigger_s. The example conceptual graph of Figure 3-9 also has examples of modified propagational relations.

The simulator handles these modified relations as follows. Whenever an activity of an action changes, the simulator compares the previous and current activity of the action. From the comparison, the simulator understands whether the action has been initiated, suspended, resumed or terminated. The simulator treats the change in activity as an event. It looks for all outgoing relations and generates the corresponding incidents. When performing this step, the simulator looks only for the presence of modified relations. If the action has been initiated, the simulator searches for relations in their base form. If the action has been suspended, the simulator looks for relations modified by an "_s." A resumed action prompts the simulator to look for relations modified by an "_r." Finally, a terminated action makes the simulator search for relations modified by an "_t."

We can verify its correct working as follows. Suppose a graph has a conceptual relation that needs to be processed when an action ends. Such a relation will be modified by an "_t" in the graph. When the action completes its execution, then the action termination state incident is processed. This is processed only after the action completes execution. At this time, the simulator processes all relations that are modified by an "_t." Therefore, the new incidents are generated only when the action completes execution, and hence are scheduled for the current simulation time. The suspension and resumption of actions can also generate incidents, and they work in a similar manner.

3.6 Executing Action Procedures

As discussed, actions are executed by calling an action procedure. This action procedure is unique to the action, and does the job of ensuring that all the essential relations for the execution are present. If any non-essential relations are present, then these relations are also taken into consideration while executing the action. The action procedure for an action does many execution-linked tasks, such as finding the associated time delay, and verifying the existence and veracity of enabling and/or disabling conditions. The procedure also creates two state incidents, the action initiation state incident and the action termination state incident. Additionally, the procedure generates value incidents due to the performance of the action, such as transferring values, reading/writing data, or adding two values. These generated value incidents will be processed at times depending on the time delay of the action.

In order to ensure that all relations associated with actions are correctly treated, we need to have a uniform method of considering all actions, irrespective of the action itself. To do this, the action procedure is written based on the schema for the action concept. As defined by [Kamath and Cyre, 1995], "a *schema* is a small conceptual graph that represents the expected context of a concept." This means that a schema of a concept contains a description of the concept in terms of its attributes.

Depending on the reason for which the schema for a concept is created, a concept may have more than one schema, and the complexity of these schemata may vary. For a given concept, we can have different schemata that exhibit different levels of detail. A relatively simple schema is sufficient for conceptual graph simulation. Such a schema includes only those relations and concepts that affect simulation.

The action procedure for performing the action is written in a particular manner. It is written using the schema for the action as a backdrop. The presence of the schema ensures that all relations that can affect the processing of the action are handled in the procedure. As discussed earlier in section 3.5, relations may be either essential or non-essential for the process of simulation. The action procedure is written so that it ensures the presence of all the essential relations for the action. The procedure further looks at all the non-essential relations that are present, and determines what needs to be done for each non-essential relation.

This process is best explained using an example. The schema for an *add* concept is presented in Figure 3-11 below. The action procedure may be broken down into six broad stages of processing. The six stages are as follows.

1) Check for enablements/disablements
2) Check for the presence of essential relations
3) Check for the presence of non-essential relations
4) Process the essential relations
5) Process the non-essential relations
6) Process the activity of the action, and the mode of the agent

Stage 1 of the procedure checks whether the action is enabled or disabled. For doing this, it searches for the *enable_if* or *disable_if* relations. If they are present, it then checks to verify that the state for the *enable_if* relation is true, and the state for the *disable_if* is false. Otherwise, the algorithm does not proceed further.

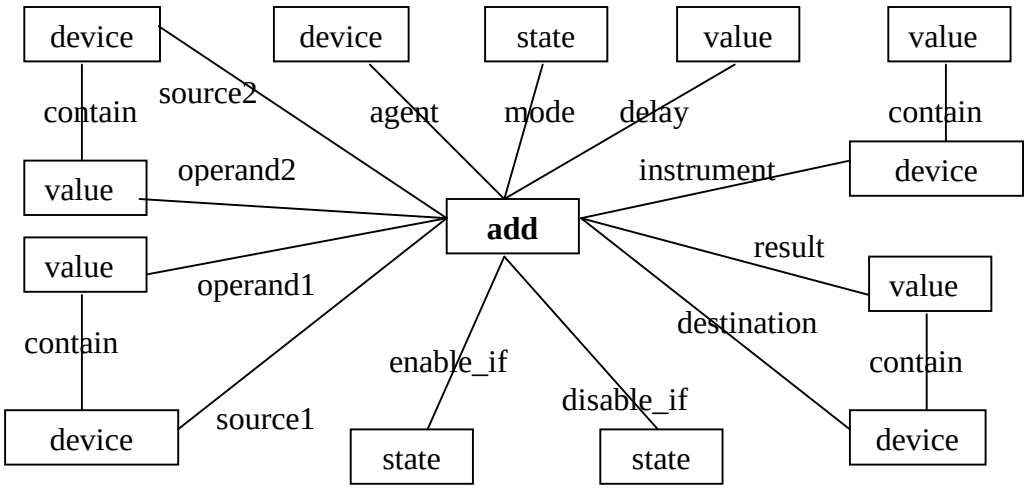

Figure 3-11–A schema for the *add* action.

Stage 2 consists of finding all the essential relations. In this case, the essential relations are the *agent*, *operand1* and *operand2* relations. The *result* relation is not essential here, as the process of addition can progress even in the absence of a *result* relation. If any of the essential relations are missing, the simulator cannot proceed, and stops simulation. It issues an error message that tells the user which essential relation is missing, and for which action. The user has to edit the graph and supply the missing essential relation, and then re-simulate the graph.

Stage 3 consists in finding all the non-essential relations that are present. Relations like *delay* and *instrument* result affect the incidents that are generated by this action. The *delay* relation is used to schedule the generated incidents at a later simulation time. The *instrument* relation necessitates the creation of an extra incident for the carrier of the action.

Stage 4 consists of processing all the essential relations that are attached to the concept. The tasks performed by this stage are unique to the action being performed. Hence, when writing the action procedure for an action, this stage is the core of the action, its actual functionality. For the addition action shown above, the two operands are fetched using the *operand1* and *operand2* relations, and added.

Stage 5 consists of processing the non-essential relations. In this stage, if the delay relation exists, the time delay of the action is taken into consideration while generating new incidents. If the instrument relation exists, an extra incident for the contents of the carrier is generated, which holds the sum of the two operands being added. The result relation is used to direct the sum of the operands to being stored in a value concept.

Stage 6 consists of changing the activity of the action. For this purpose, the mode relation of the action concept is used. Two state incidents are generated: the action initiation state incident, and the action termination state incident. These state incidents affect the state concept linked to the action concept using the mode relation. They change the activity from inactive to active, and then back to inactive after the time delay of the action. Also, the mode of the agent, the concept that performs the action, is set to busy for a period of time equal to the time delay of the action. At the end of this time, the mode of the agent concept is changed back to ready.

Algorithms for creating the action procedures for some commonly occurring action words–transfer, read, write, execute, increment, clear, request and grant–are given in Appendix C.

3.7 Expanding Device Concepts for Simulatability

As discussed earlier, the non-simulatable device concept type is used to describe physical devices, such as processors, memory, registers and so on. Concepts of this type tend to play a passive role in simulation. The three primary roles played by device concepts in a conceptual graph are to store values, affect the enablement or disablement of events and actions by virtue of their operational modes, and act as agents for executing actions.

One improvement that will greatly ease the burden of building a simulatable conceptual graph is the use of *schema*. A *schema* is a small conceptual graph that represents the expected context of a concept [Kamath and Cyre, 1995]. This means that a schema of a concept contains a description of the concept in terms of its attributes. For example, a schema for a read-only memory (ROM) may have the following attributes: the number of bits used to represent a word, the number of words present in the ROM, and the values of these words. A schema for an I/O controller may have its operational mode and a buffer for holding data. A schema for an increment action may contain a list of conceptual relations that affect the execution of the action.

Let us consider a situation when a device concept for register occurs in the graph that has to be simulated. When the register device concept is encountered, during graph preparation for simulation (see section 3.2), we need to use the schema for a register concept. This schema consists of a device concept and a value concept, related to each other by the *contain* relation. Such a schema for the register is seen in Figure 3-12.

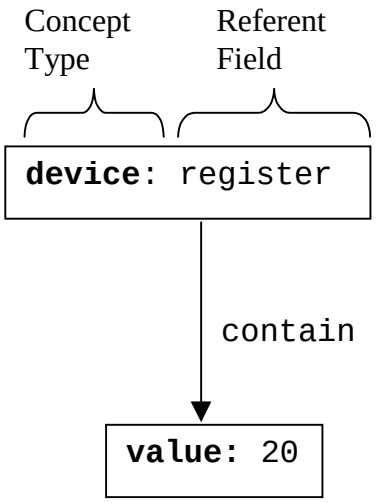

Figure 3-12–A schema for the register concept.

Another example of a conceptual graph utilizing device concepts is shown in Figure 3-13. This figure represents the English sentence “When the I/O controller is ready, the processor transfers 1024 bytes of data from the I/O controller to memory.” The

graph in Figure 3-13 is not completely simulatable. Certain relations that are necessary for simulation are not present in the graph. These relations have not been added to Figure 3-13, as they do not play a part in illustrating the role of device concepts in simulation.

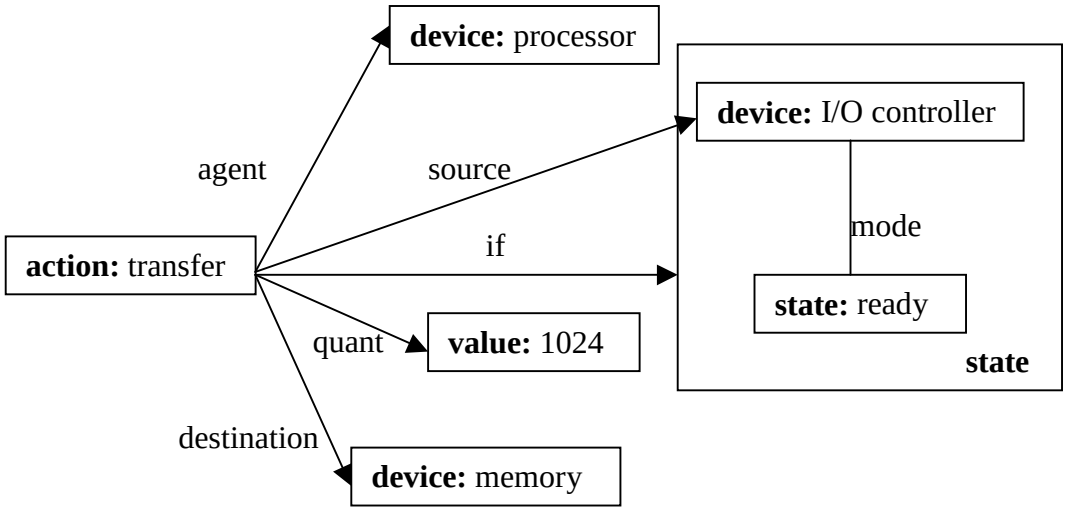

Figure 3-13–Another conceptual graph using device concepts.

When we simulate the graph in Figure 3-13, we see that an enabling condition exists. To check the veracity of the enabling condition, we first simulate the inner graph that describes the enabling condition. This inner graph checks whether the input/output (I/O) controller is ready. The *mode* relation indicates that the I/O controller has a certain mode of operation.

However, what the graph implies is, we need to check the operational mode of the I/O controller and see if it is ready. Some devices have an associated operational mode. This operational mode of the device concept is a state concept that holds the status of the device–such as busy, reset, disabled, and so on. In Figure 3-13, this state concept is not yet associated with the device concept. Therefore, devices such as I/O controllers and processors need to be related to a state concept that holds the device's operational mode. Figure 3-14 shows a schema for the I/O controller with its operational mode. The mode of operation of the I/O controller is linked using the *mode* relation.

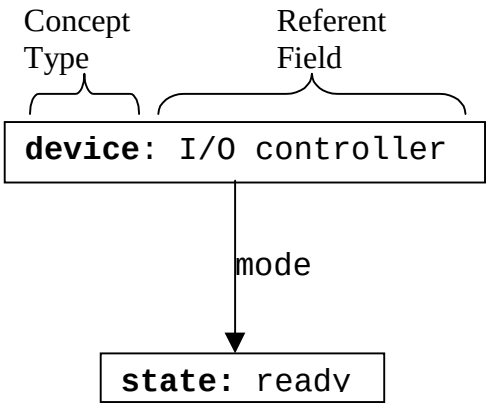

Figure 3-14–The I/O controller and it's associated operational mode.

Proceeding with the simulation of the graph in Figure 3-13, let us assume that the enabling condition is true. We need to perform the transfer of 1024 values from the I/O controller to the memory concept. This action entails that both the I/O controller and the memory have to be associated with at least 1024 values, so that the transfer may successfully be simulated. The process of associating multiple values with a device concept is explained below.

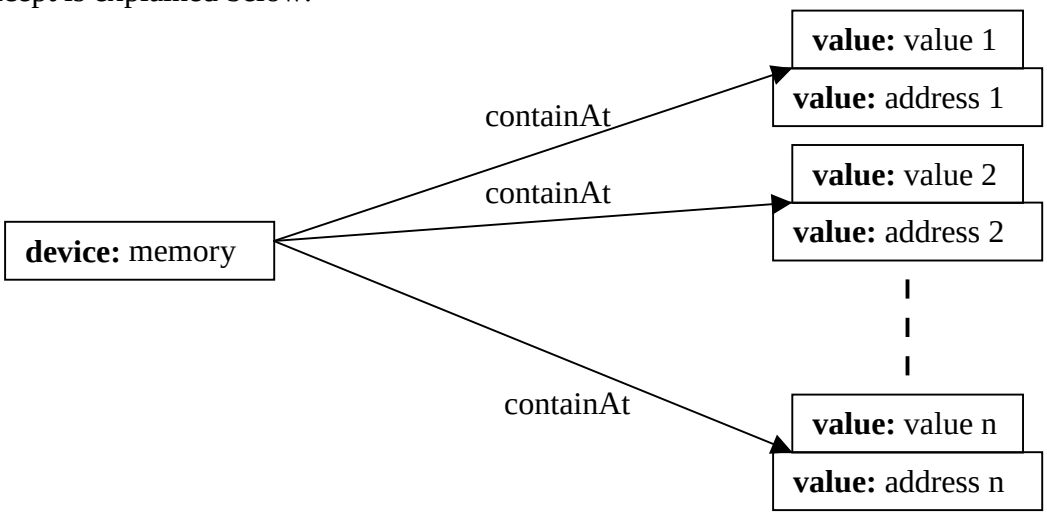

Figure 3-15–A schema for memory.

When multiple values have to be associated with a device concept, we may need to refer to each of those values individually. To achieve this, each value associated with a device is also given a unique address. This address helps in referring to a value individually. Each (value, address) pair is now related to the device concept using a new

containAt relation. This relation implies that the device concept now "contains" the value (stored in the value field) "at" the given address (stored in the address field). Figure 3-15 shows an example conceptual graph of a memory device concept that is associated with 'n' values, using the newly defined *containAt* relation.

Figure 3-13 thus illustrates all the three primary functions played by device concepts. The memory and the I/O controller store values, the operational mode of the I/O controller acts as an enabling condition for the execution of the transfer, and the processor functions as an agent performing the transfer action.

With respect to device concepts, schemata are created for a set of commonly occurring devices. These created schemata are then stored in a schema database. When a concept in the graph being simulated needs to be elaborated using schemata, a suitable schema for that concept is picked from the schema database. The concept in the graph being simulated is elaborated with the picked schema by joining (merging) the concept being elaborated with the head concept of the schema. This may require restricting (specializing) the head concept to match the graph's concept. The joining of the schema may be extended by restricting and joining related concepts of the schema in the graph being simulated, as discussed in section 2.1.

This process of device expansion is better explained using an example. Consider that the device concept for memory in Figure 3-13 needs to be elaborated using the example schema for memory shown in Figure 3-15. The device concept for memory (in Figure 3-13) is joined (merged) with the device concept in Figure 3-15. Suppose the device concept for memory in Figure 3-13 is a special type of memory–say random access memory (RAM). But the schema in Figure 3-15 is a graph that is true for any type of memory. Then, the merging of the two device concepts requires that the device concept of Figure 3-15 be restricted (specialized) to match the RAM-type of the device concept in Figure 3-13. By integrating a schema for memory into the graph, we now have a more complete picture of how the entire system looks like. A more thorough treatment of elaborating conceptual models with schemata is found in [Kamath and Cyre, 1995].

3.8 Summary

In this chapter, the design for the simulator was developed. The different types of concepts were discussed, and their simulatability determined. The means for making each type of concept simulatable were presented. The method of using queues to perform simulation was considered. Processing for each type of incident was explained in detail. Conceptual relations were classified into executional and propagational relations, and each type was explained. Executing action procedures and expanding device concepts for simulatability were also discussed. We next consider the implementation of the simulator.

Chapter 4 Implementation

This chapter describes the implementation of the simulator in Java. First, a high level model of the system using Unified Modeling Language (UML) tools is presented. This helps to document the software architecture of the simulator. Next, the standards and external software packages used to implement the simulator are also discussed briefly. These include the proposed Conceptual Graph Interchange Format (CGIF) and the Notio API for dealing with conceptual graphs represented in CGIF.

4.1 UML Model of the Simulator

The creation of a high level model is an important software design step, as it allows the software developer to organize the software into its constituent classes. These classes may then be used to implement different sections of the simulator. A high-level model of the conceptual graph simulator using the Unified Modeling Language (UML) is shown below in Figure 4-1. The figure was generated using Rational Rose /C++ Demo version 4.0.3 [Rational Software Corp., 2001]. Figure 4-1 is a class diagram, which shows the different classes created for the implementation of the simulator.

CGSimulator is the main class. It reads the conceptual graph from a text file using the FileInput class, creates initial incidents using the Incident class, performs the simulation using the SimulationEngine class, and logs the results of the simulation to a text file using the FileOutput class.

The FileInput and FileOutput classes contain static methods, and hence do not need to be instantiated before use. The FileInput class accepts a parameter that holds the name of the input text file. The FileOutput class opens a log file with the name “cgsimlog.txt” and logs the date and time of the current simulation. It also writes all the results of the simulation to the log file mentioned above.

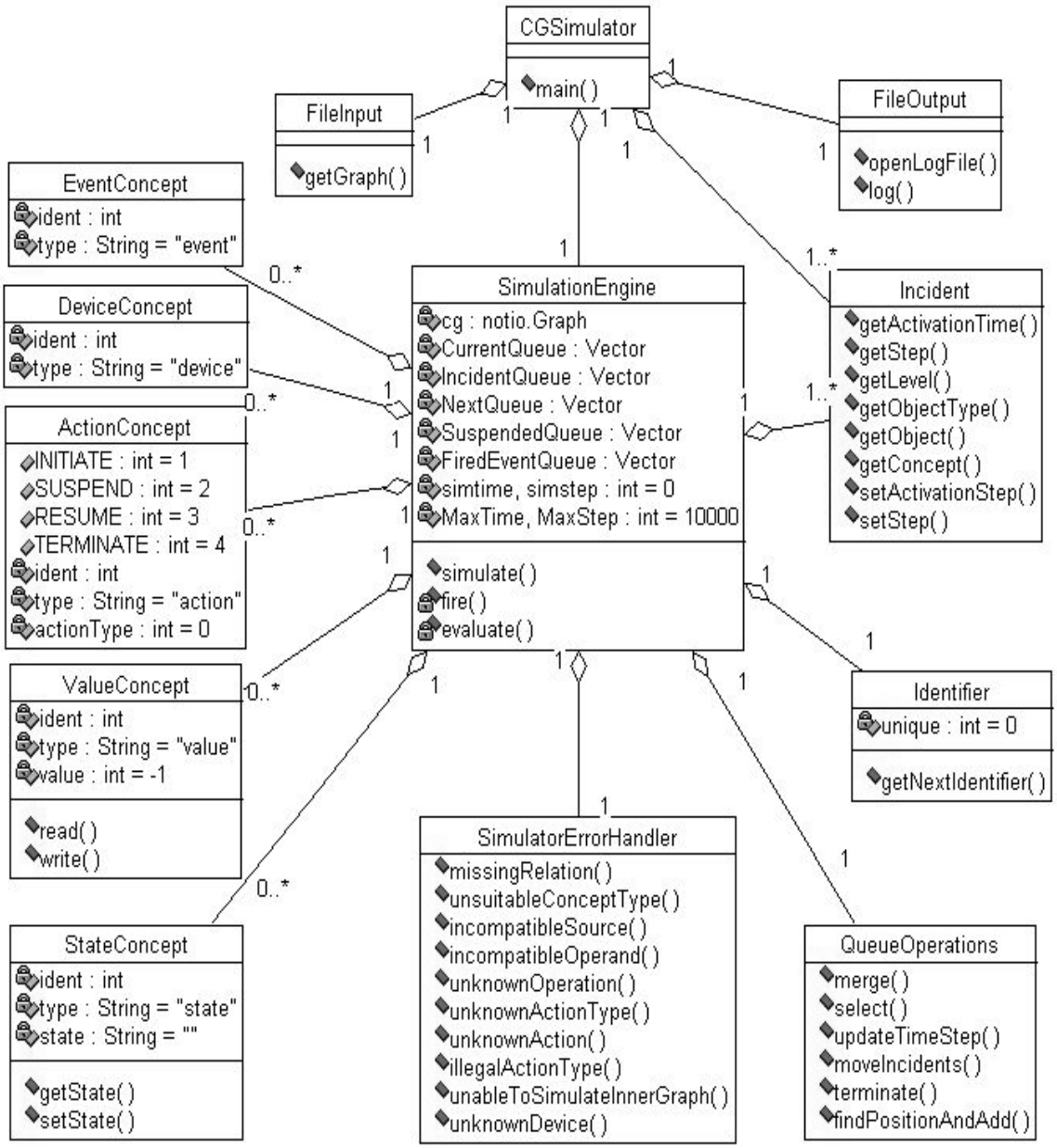

Figure 4-1–A UML model of the CG simulator.

The SimulationEngine class is the most important class of the simulator. Only the attributes of the class are shown in Figure 4-1, the operations of the class are not shown. The SimulationEngine class has twelve operations; some of these are discussed later. The SimulationEngine class uses four main classes: the Incident class, the Identifier class, the QueueOperations class, and the SimulatorErrorHandler class.

The Incident class is used to represent an incident, which has been defined in section 3.1. The class has six attributes: the activation time, the activation step, the level of nesting, the type of incident, the concept which this incident represents, and the causing concept for this incident. Operations of this class include methods to access each of these attributes, and methods to change the activation time and step.

As discussed in section 3.2, the conceptual graph needs to be prepared before simulation. The Identifier class is a class that supplies unique identifiers when the graph is prepared (concepts are added to the graph). The reason for making this class (and not an integer within the SimulationEngine class) is to allow for different classes to perform different sections of preparing the conceptual graph.

The purpose of creating the QueueOperations class was to simplify the handling of queues. As explained in section 3.3, the simulation of the graph is done using queues. Operations such as merging of two queues, moving incidents from one queue to another, and so on, are very frequently performed. Therefore, this class stores a frequently used set of queue operations as procedures. These procedures are called with proper parameters to perform each queue operation. The procedures of this class are as follows:

1) the merging of two queues
2) the selection of incidents from one queue to another based on current simulation time
3) the movement of incidents from one queue to another based on a causing concept
4) the removal of incidents from a queue based on a causing concept, and
5) the addition of a new concept at the correct position in a time-ordered queue

The SimulatorErrorHandler class was created for the treatment of errors that may occur during simulation. This class contains a set of procedures, where each procedure concerns a specific error. These procedures log error messages to the output file using the FileOutput class. Each procedure accepts parameters; this allows the simulator to indicate the module in which the error occurred. In addition, each procedure is overloaded to allow the user to change the severity of an error. There are three levels of severity: error, warning, and note. A severity level of “error” means that the simulation is aborted, as it is

unable to continue. A severity level of “warning” means that the graph lacks information. In this case, simulation is able to continue, but it is advisable to include this information for proper simulation. A severity level of “note” is used to inform the user of something unusual, but in no way affecting the process of simulation. The procedures are as follows:

1) Missing relation–a conceptual relation is missing from the graph
2) Unsuitable concept type–an argument of a relation does not match the concept type that the relation expects
3) Incompatible source–source of an action does not match the action being performed
4) Incompatible operand–the operand of an action does not match the action being performed
5) Unknown operation–the action has to perform an unknown operation
6) Unknown action type–an unknown action type is encountered (other than initiate, suspend, resume, and terminate)
7) Unknown action–an action that does not have an associated procedure is encountered
8) Illegal action type–an illegal action type is encountered (for example, if an action is not running, and it is terminated, or suspended, that is an error)
9) Unable to simulate inner graph–an inner graph (of a state concept) could not be properly simulated
10) Unknown device–a device for which no schema graph exists is encountered

The SimulationEngine class also uses five other classes, which are specialized classes for each type of concept. These five classes are EventConcept, DeviceConcept, ActionConcept, ValueConcept, and StateConcept. These specialized classes are used for holding information specific to a concept type. As an example, the ActionConcept class is used exclusively for action concepts. This class has an attribute for the activity of an action (initiated, suspended, resumed or terminated). Similarly, the ValueConcept class has an attribute for holding a value. These classes play an important role in simulation; they are used in the generation of new incidents of their respective types. In addition, schema graphs for device concepts such as memory and register utilize these classes instead of the parent Concept class.

Each of these five classes contains attributes for an identifier and a concept type. The common methods for these classes are operations for accessing the identifier and the concept type for each class. Apart from this, each class has one or more attributes specific to its concept type, and operations to access and/or modify these attributes. A more detailed description of each class is available in Appendix D, where the UML documentation for each class is listed.

We shall now take a look at the standards and software packages used for the implementation of the conceptual graph simulator.

4.2 Standards and Software Packages Used

The graphic way of representing conceptual graphs (used until now) is to use rectangles for concepts and arcs for relations. This form of representation is both easy to understand and intuitive. However, when such a graph is the input to the simulator, the graphic form is cumbersome to handle. As an alternative, a new textual representation for conceptual graphs has been proposed. This new representation is the Conceptual Graph Interchange Format (CGIF); it is still not an ISO standard, but a working draft of the proposed ISO standard is available at [Sowa, 2001a]. An earlier draft has been published [Sowa, 1999]. [Sowa, 2001b] has more examples on CGIF and illustrative representations of conceptual graphs.

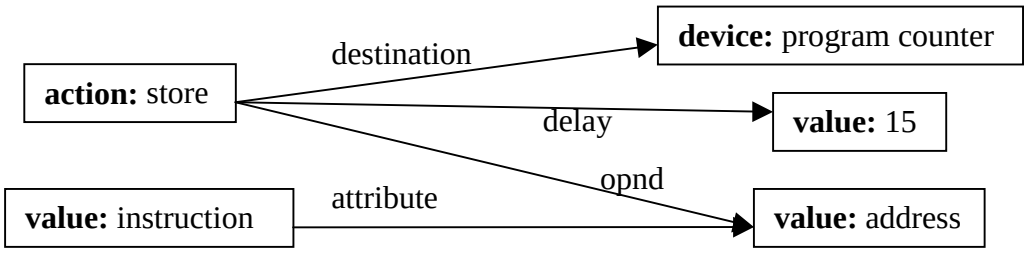

Figure 4-2–An example conceptual graph in graphic form.

Let us see an example of a conceptual graph represented in CGIF. Consider Figure 4-2, which depicts the English sentence “The address of the instruction is stored in the program counter.” When the graph in Figure 4-2 is represented using CGIF, the resulting graph is seen in Figure 4-3.

```
[action:*a ‘store’]
[value:*b ‘15’]
[device:*c ‘program counter’]
[value:*d ‘address’]
[value:*e ‘instruction’]
(destination ?a ?c)
(opnd ?a ?d)
(delay ?a ?b)
(attribute ?e ?d)
```

Figure 4-3–The conceptual graph of Figure 4-2 in CGIF.

We next discuss Notio, an Application Programming Interface (API) specification for a set of Java classes, designed to provide an implementation-independent interface for manipulating conceptual graphs. The design of the Notio API is described in [Southey and Linders, 1999]. The specification is basically a set of guidelines to develop an API for manipulating conceptual graphs. This specification lays importance on the fact that the API is implementation-independent. Implementation-independence of the API implies that they should be able to handle multiple textual representation formats of conceptual graphs, and yet provide a single set of classes to an application developer.

[Southey, 2000] provides a reference implementation of the Notio API. This reference implementation is a set of Java classes, which is able to parse both CGIF and linear-form (LF) representations of conceptual graphs. The API provides classes for graphs, concepts, concept types, relations, relation types, and so on. The API also provides a means of building conceptual graphs applications–like the conceptual graph simulator under discussion. Other software used for implementing the simulator was the Java™ version 1.3.0, with the Java™ 2 Runtime Environment.

4.3 User Interface of the Simulator

The user interface for the graph simulator is described here. At present, the user interface is still command-line based, that is, the user has to issue a command at the prompt to simulate the graph. The simulator offers a debug feature that allows the user to step through each cycle of the simulation. Also, the simulator offers three output options, each varying in the amount of detail that the simulator logs to the output file. The simulator assumes that the graph to be simulated is stored in a file. It also assumes that the graph has been represented using CGIF. Finally, the file to be simulated is assumed to be the same folder in which the simulator is present.

In the normal mode of operation, the simulator is invoked by typing the *run* command at the command prompt and supplying the name of the file to be simulated. The results of the simulation are automatically logged to a text file named "cgsimlog.txt". The simulator allows the user to step through the simulation. This feature is called the debug mode of operation. This feature simulates the graph one cycle at a time, and then allows the user to examine the results. When the user has finished examining the results, he may hit the "enter" key to simulate one more cycle. The debug mode of operation is activated by typing "-d" after the *run* command and the file name.

An additional feature that has been implemented allows the user to determine the amount of output generated by the simulator. Varying the output-option parameter controls the amount of output that the simulator writes to the file. The simulator offers three output-options: less, medium, and more. Typing one of these output-options after the *run* command and the file name will vary the amount of detail that is logged by the simulator to the output file. The user may vary the output-option both in the normal and debug modes of operation. Appendix A provides more detail on the simulator usage.

Chapter 5 Experiments and Results

This chapter of the document discusses the experiments carried out to validate the working of the simulator. We first look at the domain of interest, that is, the particular area in which the test cases are concentrated, and the reason for limiting the domain. We then examine possible tools to create the test cases. The next section covers a series of test cases to ensure that the simulator successfully simulates conceptual graphs. The final section covers test cases that verify the simulator's ability to pick up errors in the graphs and bring them to the user's attention. All test cases and their results are included in Appendix E of the document.

5.1 Domain of Interest

The domain of interest for this document covers a set of 130 US patents issued on Direct Memory Access (DMA) systems. This domain was chosen as a particularly good domain for experimentation, because of the following two reasons.

First, the normal usage of the English language tends to exhibit an extraordinarily large vocabulary. Also, non-technical documents exhibit the presence of slang words, whose meanings may not be documented. For the simulator to perform well on a set of test cases, it requires a knowledge base that contains the meanings of the words in the test cases. When non-technical documents are used to create test cases, the size of the dictionary needed to hold all the required words and their meanings is very large. By restricting the domain of interest to a narrower, more technical domain, we can reduce the vocabulary needed and work with smaller dictionary files.

Second, within written English, sentence construction and grammar vary from person to person. These variations in the style of writing are more pronounced when authors come from diverse geographical and language backgrounds. Test cases based on

a broad variety of writing need a large set of rules in order to be understood by the software. But the authors of technical documents, such as patents, tend to follow more regular patterns of sentence formation. By restricting the domain to a more technical one, the set of rules needed to understand the sentences is reduced. This reduction in rules makes it easier for generating conceptual graphs from these sentences.

Due to the above reasons, the domain of US patents issued with respect to DMA systems was chosen as the current domain of interest. The test cases for testing the simulator were generated from these patents.

5.2 Generation of Test Cases

The test cases for the simulator are based on the set of 130 US patents on DMA systems. For this document, the test cases are a set of conceptual graphs that have to be simulated. The conceptual graphs were generated as follows. For each conceptual graph, a set of sentences was chosen from these patents. These sentences were then manually converted to conceptual graphs and stored in the CGIF notation. Smaller test cases, involving ten or fewer concepts and relations, were hand-coded; that is, the sentences were directly converted to CGIF and stored in a file.

When dealing with larger sets of sentences, keeping track of all concepts and relations became a tedious job. Hence, the CharGer conceptual graph editor was used to model the sentences in graphical form, and then store the graphs using CGIF. The CharGer conceptual graph editor presents an easier means of generating conceptual graphs. The editor provides an easy-to-use, graphical approach to creating conceptual graphs. Both concepts and relations are represented graphically. The means for creating hierarchical conceptual graphs has also been implemented in this editor. One point to be noted is that the test graphs shown below all depict the graph before simulation. They do not show the additional concepts and relations added to make the graph simulatable. More information about the editor is available in [Delugach, 2000].

5.3 Analysis of Simulator Output for Simple Test Cases

This section briefly describes some simple test cases, and an explanation of the results. This section looks at three test graphs that simulate correctly, and explains the simulator output. It also looks at a test graph that does not simulate fully, and describes how the simulator reacts to this graph.

The actual testing of the simulator covered twenty-two test cases, which are presented in Appendix E of this document. Of the twenty-two, the first twelve test cases are test cases for checking functionality. These are test cases that check for the correct working of the simulator. Each of these test cases will check for one particular design aspect of the simulator. For each of these test cases, we will first describe which aspect the test case is designed to test, the expected output, and the observed output. The next ten test cases are for verifying error handling. These are test cases that should not simulate successfully. In these test cases, the simulator must abort simulation with an understandable error message. The message should provide enough information to allow the user to locate the error in the graph.

The first test case we look at performs the transfer of data from one register to another using a bus as the instrument for the transfer. This test case first loads one register with the number “17”, the second register with the number “23”, and then transfers the value in the first register to the second register. The system bus is used as the instrument for all the three operations. All the operations have time duration of 10 time units. The test case is shown in Figure 5-1 in graphic form, and also shown in Figure 5-2 in the CGIF notation. The expected result of simulating this graph is that the first register gets loaded with the number “17” at time 10, the second register gets loaded with the number “23” at time 20, and then second register gets over-written by the number “17” at time 30. The system bus should reflect each of these operations.

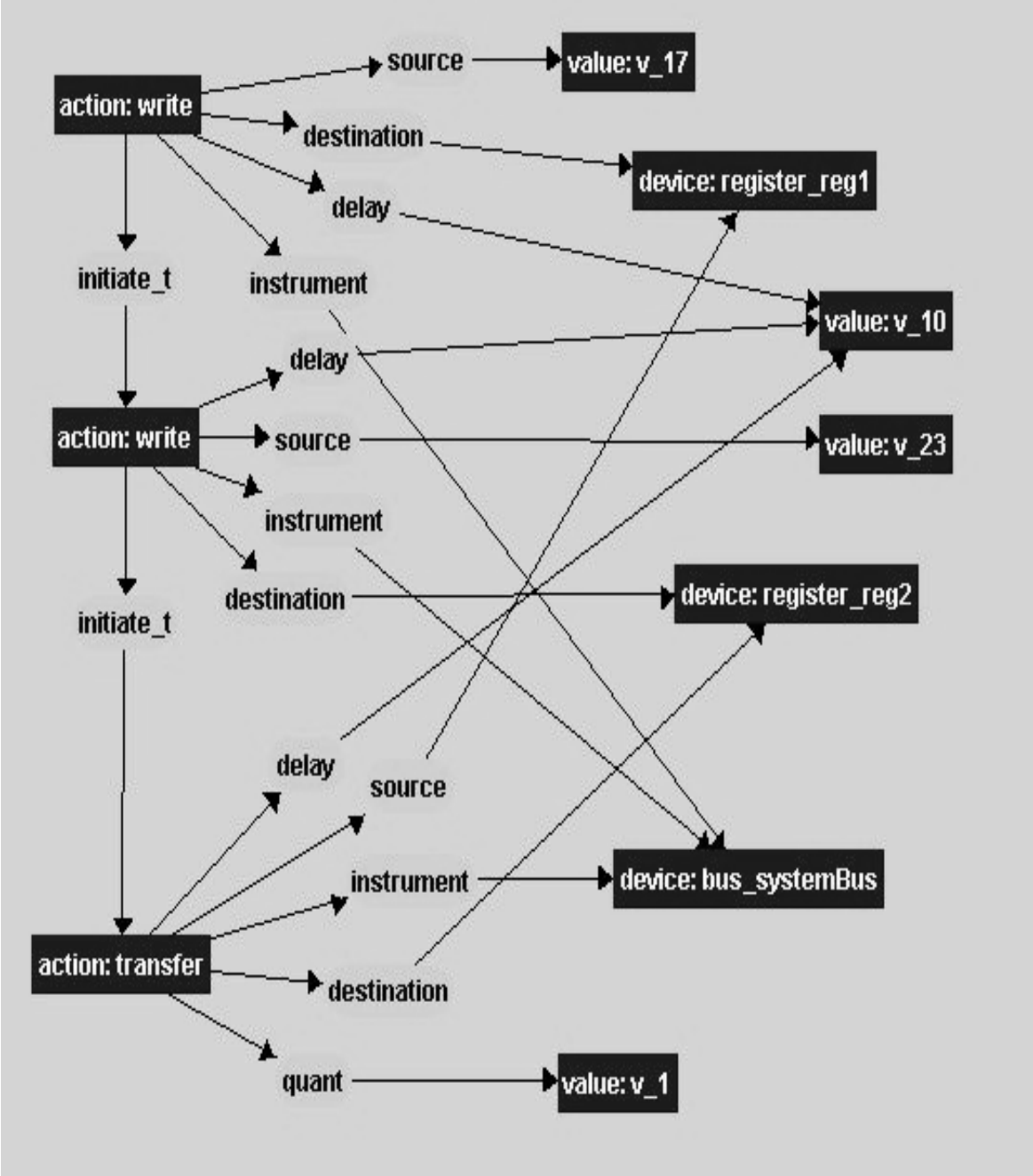

Figure 5-1–First test case in graphic form.

The output of the simulator for this test graph is shown in Figure 5-3 below. As seen from the output, the first register is loaded with “17” at time 10, the second register is loaded with “23” at time 20, and the value of “17” is transferred from the first register to the second register at time 30. The system bus reflects all these operations.

```
[Proposition:''
[action:*a'write']
[device:*b'register_reg1']
[value:*c'v_17']
[value:*d'v_10']
[action:*e'write']
[device:*f'register_reg2']
[value:*g'v_23']
[action:*h'transfer']
[value:*i'v_1']
[device:*j'bus_systemBus']
(source?a?c)(destination?a?b)(delay?a?d)(instrument?a?j)
(initiate_t?a?e)
(source?e?g)(destination?e?f)(delay?e?d)(instrument?e?j)
(initiate_t?e?h)
(source?h?b)(destination?h?f)(delay?h?d)(quant?h?i)
(instrument?h?j)]
```

Figure 5-2–First test case in CGIF notation.

```
Thu Apr 26 19:52:03 EDT 2001
File simulated: transferregbus.cgf
Graph has 10 concepts and 15 relations.

TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: write (mode = active)
TIME: [10]
      STEP: [0]
            DEVICE: register_reg1 (contain = '17')
            DEVICE: bus_systemBus (contain = '17')
            ACTION: write (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: write (mode = active)
TIME: [20]
      STEP: [0]
            DEVICE: register_reg2 (contain = '23')
            DEVICE: bus_systemBus (contain = '23')
            ACTION: write (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: transfer (mode = active)
TIME: [30]
      STEP: [0]
            DEVICE: register_reg2 (contain = '17')
            DEVICE: bus_systemBus (contain = '17')
            ACTION: transfer (mode = inactive)

FILE SIMULATION DONE!
```

Figure 5-3–Simulator output for first test case.

The second test case we shall look at performs the initiation of a WAKE action that has an enabling condition. The WAKE action is initiated by the SLEEP action only if the enabling state is true. The enabling state is represented by the condition "either the SLEEP action is active, or the value 2344 is greater than the value 12433." Figure 5-4 shows the second test case in graphic form, and Figure 5-5 shows it in CGIF notation.

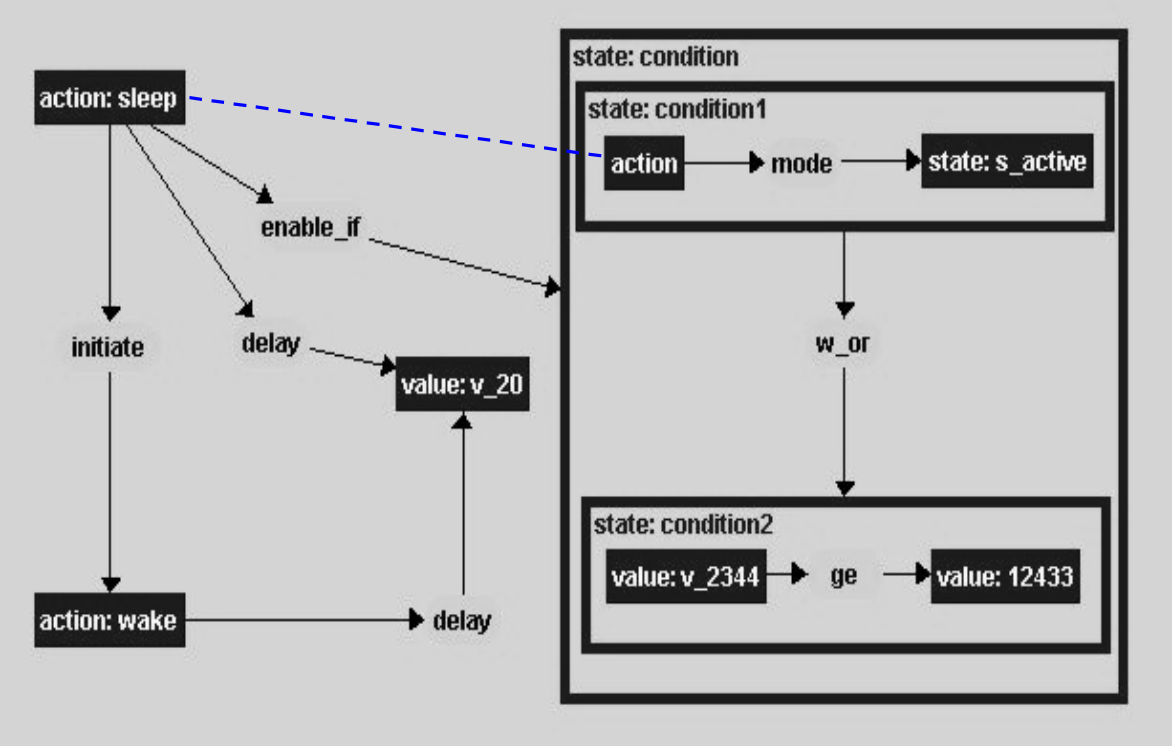

Figure 5-4–Second test case in graphic form.

```
[Proposition:''
[action:*a'sleep']
[action:*b'wake']
[value:*c'v_20']
[state:*d's_false'
      [state:*e'condition1'
            [action:?a]
            [state:*f's_active']
            (eq?a?f)]
      [state:*g'condition2'
            [value:*h'v_2344']
            [value:*i'v_12433']
            (ge?h?i)]
      (w_or?e?g)]
(enable_if?a?d)(initiate?a?b)(delay?a?c)(delay?b?c)]
```

Figure 5-5–Second test case in CGIF notation.

The expected result of this action should be that the WAKE action is initiated. This is because the enabling condition is true. Though 2344 is not greater than 12433, the SLEEP action is active, and since the condition is the “or” of the two conditions, the enabling condition is true. The objective of this test case is to show that the simulator allows for the presence of nested graphs, and evaluates the nested graph for its veracity.

```
Thu Apr 26 19:53:55 EDT 2001
File simulated: states.cgf
Graph has 4 concepts and 4 relations.

TIME: [0]
      STEP: [1]
            ACTION: sleep (mode = active)
                  processing an AND or OR relation in evaluate
                  processing an EQ or NE relation in evaluate
                        The first concept is an action concept.
                        Its mode is: active
                        The second concept is a state concept.
                        Its state is: active
                  First value is: 'true'
                  processing an arithmetic compare in evaluate
                        First value is: 2344
                              Second value is: 12433
                  Second value is: 'false'
                  The final result is: true
            sleep initiates action WAKE
      STEP: [3]
            ACTION: wake (mode = active)
TIME: [20]
      STEP: [0]
            ACTION: sleep (mode = inactive)
                  processing an AND or OR relation in evaluate
                  processing an EQ or NE relation in evaluate
                        The first concept is an action concept.
                        Its mode is: inactive
                        The second concept is a state concept.
                        Its state is: active
                  First value is: 'false'
                  processing an arithmetic compare in evaluate
                        First value is: 2344
                        Second value is: 12433
                  Second value is: 'false'
                  The final result is: false
            enable condition exists, action not enabled, returning
            ACTION: wake (mode = inactive)
FILE SIMULATION DONE!
```

Figure 5-6–Simulator output for second test case (edited).

As we need to get inside the working of the simulator, the “medium” output-option has been used to generate the output. An edited form of the output, showing only the simulation results, and indented for readability, is shown in Figure 5-6. The entire output of the simulator is available in Appendix E as test case 10. As we see from the output, the SLEEP action is made active. The simulator then checks the state for its veracity. It finds that the state is a nested graph, and it also finds that it is testing for a logical “and” or “or” condition. It then checks the operands for the logical “or”, and finds that, in turn, they are nested graphs. The simulator then evaluates each individual operand. The first operand is that the SLEEP action is active, and this is true. Therefore the first value for the “or” is true. The second operand is that 2344 is greater than 12433 and this is false. Therefore the second operand for the “or” is false. The logical “or” of true and false is true. Hence the condition is true, and the WAKE action is enabled. This state evaluation is again repeated when the SLEEP action terminates, to account for relations that have to be processed when the action terminates.

The third test case we shall look at is a model of an iterative action using the *initiate_t* relation and an enabling condition. The test setup initializes a value concept to a value of “1”. This value is incremented until it reaches a value of “7”. At this point counting is stopped. The objective of this test case is to show that high-level constructs such as “for-loops” can be easily modeled using conceptual graphs. The third test case is shown in Figure 5-7 in the CGIF notation and Figure 5-8 in graphic form.

```
[Proposition:''
[action:*a'write']
[value:*b'v_10']
[value:*c'v_1']
[value:*d'data']
[state:*e''
     [value:*f'v_7'][value:?d''](lt?d?f)]
[action:*g'increment']

(source?a?c)(destination?a?d)(delay?a?b)(initiate_t?a?g)
(initiate_t?g?g)(delay?g?b)(operand?g?d)(result?g?d)
(enable_if?g?e)]
```

Figure 5-7–Third test case in CGIF notation.

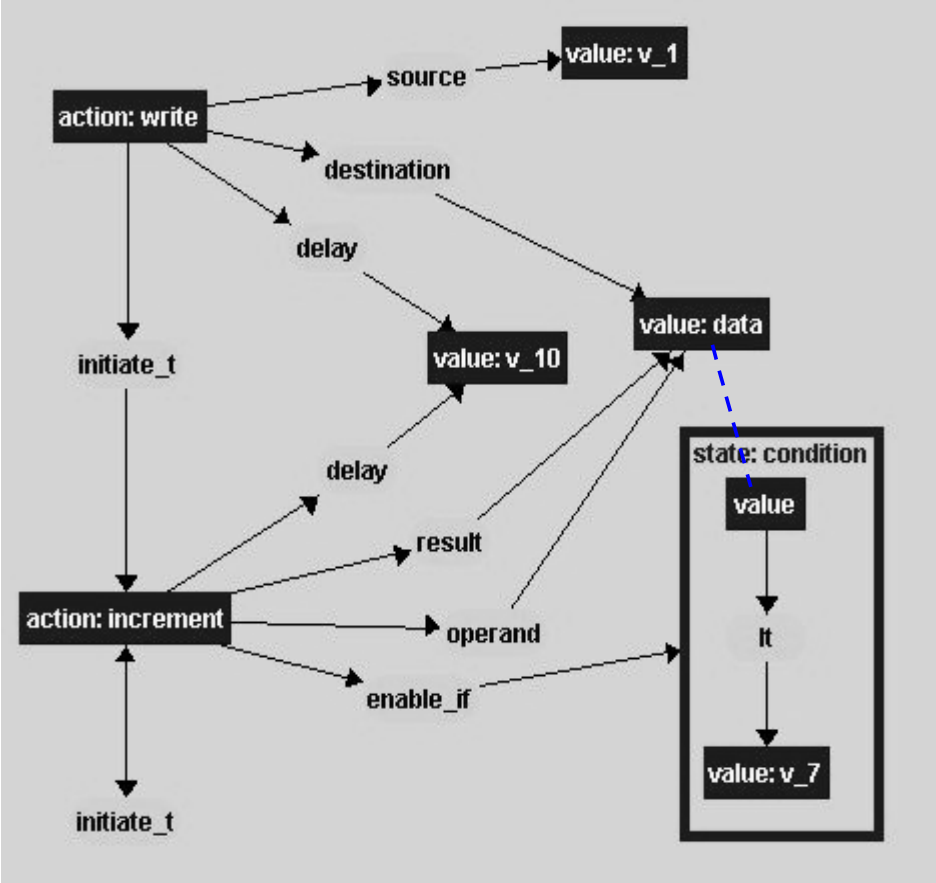

Figure 5-8–Third test case in graphic form.

The expected result of simulating this test case is as follows. The value concept labeled “data” should be initialized with a value of “1” at time 10. Thereafter, at successive time increments of 10, the value concept should get incremented until it reaches a value of “7”. At this point, the enabling condition will be false, so the value concept should not increment further. The simulation should come to an end at this point. The results of the simulation of the third test case are shown in Figure 5-9. As seen from the results, the simulator output follows the expected results.

We have looked at three test cases where the simulator has functioned correctly. The output of the simulator has matched the expected output for each of these three test cases. More test cases that check the functionality of the simulator are covered in test cases one through twelve of Appendix E. The simulator should also be able to identify errors in test graphs and notify the user about these errors. This is considered next.

```
Thu Apr 26 19:58:24 EDT 2001
File simulated: valueexample.cgf
Graph has 6 concepts and 9 relations.

TIME: [0]
      STEP: [1]
            ACTION: write (mode = active)
TIME: [10]
      STEP: [0]
            VALUE: data (eval = '1')
            ACTION: write (mode = inactive)
      STEP: [2]
            ACTION: increment (mode = active)
TIME: [20]
      STEP: [0]
            VALUE: data (eval = '2')
            ACTION: increment (mode = inactive)
      STEP: [2]
            ACTION: increment (mode = active)
TIME: [30]
      STEP: [0]
            VALUE: data (eval = '3')
            ACTION: increment (mode = inactive)
      STEP: [2]
            ACTION: increment (mode = active)
TIME: [40]
      STEP: [0]
            VALUE: data (eval = '4')
            ACTION: increment (mode = inactive)
      STEP: [2]
            ACTION: increment (mode = active)
TIME: [50]
      STEP: [0]
            VALUE: data (eval = '5')
            ACTION: increment (mode = inactive)
      STEP: [2]
            ACTION: increment (mode = active)
TIME: [60]
      STEP: [0]
            VALUE: data (eval = '6')
            ACTION: increment (mode = inactive)
      STEP: [2]
            ACTION: increment (mode = active)
TIME: [70]
      STEP: [0]
            VALUE: data (eval = '7')
            ACTION: increment (mode = inactive)

FILE SIMULATION DONE!
```

Figure 5-9–Simulator output for third test case.

The fourth test case contains a logical error. The TRANSFER action tries to suspend the SLEEP action, which has not been initiated. The objective of this test case is to check whether the simulator can detect that the SLEEP action cannot be suspended, as it has not been initiated in the first place. The expected output for this test case is that the TRANSFER action should execute successfully, but when it tries to suspend the SLEEP action, the simulator should abort simulation, giving the user a suitable error message. The test graph for the fourth test case is shown in Figure 5-10 in graphic form, and in Figure 5-11 in CGIF notation.

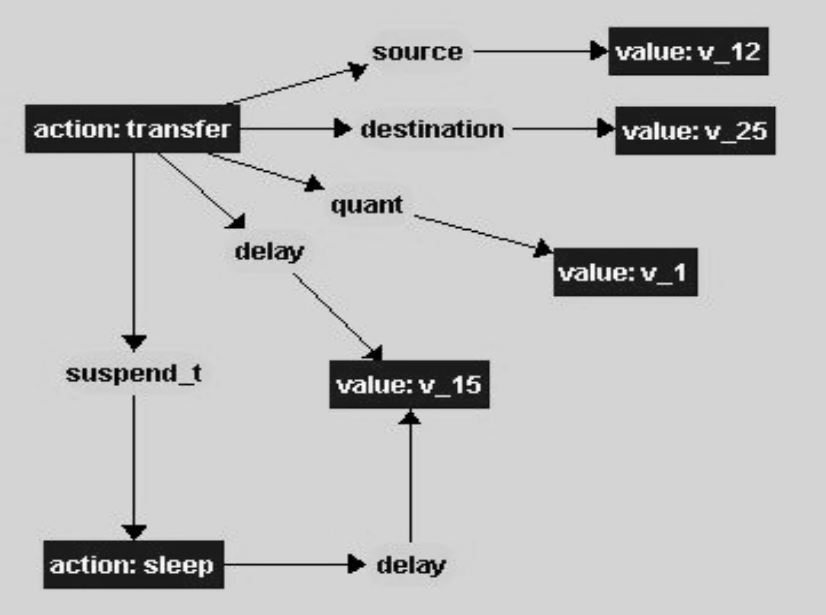

Figure 5-10–Fourth test case in graphic form.

```
[Proposition:''
[action:*a'transfer']
[value:*b'v_12']
[value:*c'v_25']
[value:*d'v_15']
[value:*e'v_1']
(source?a?b)
(destination?a?c)
(delay?a?d)
(quant?a?e)
[action:*f'sleep']
(delay?f?d)
(suspend_t?a?f)]
```

Figure 5-11–Fourth test case in CGIF notation.

The results of simulating the fourth test case are shown in Figure 5-12. As we see from the output, the TRANSFER action executes successfully, but the simulator aborts simulation when the SLEEP action has to be suspended. It also notifies the user that the SLEEP action cannot be suspended, as it is not currently active. This test case is also included in Appendix E as test case 16.

```
Thu Apr 26 20:01:04 EDT 2001
File simulated: illegalsuspend.cgf

Graph has 6 concepts and 6 relations.

TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: transfer (mode = active)

TIME: [15]
      STEP: [0]
            VALUE: v_25 (eval = '12')
            ACTION: transfer (mode = inactive)
      STEP: [1]
ERROR : Illegal action type: cannot SUSPEND action 'sleep' as it
is not ACTIVE. SIMULATION ABORTED!

FILE SIMULATION DONE!
```

Figure 5-12–Simulator output for the fourth test case.

We have looked at four sample test cases. These test cases show that the simulator correctly performs the simulation of completely specified conceptual graphs. It also helps in analyzing the incompletely specified graphs by giving the user suitable error messages that help the user trace the source of the error in the graph and correct it.

We now look at the performance of the simulator on a larger conceptual graph that was manually created from a patent on DMA systems. We explain how the simulator proves useful in validating such conceptual graphs.

5.4 Performance of the Simulator on a Large Conceptual Graph

The example conceptual graph in this section was manually from a patent on DMA systems. The English sentences that were chosen for modeling are as follows. “First the processor sends a command to a peripheral i/o device to transfer data. Next the DMA controller requests the bus. When the processor grants the system bus, the DMA controller transfers the data when it is ready. When the transfer between the device and memory is completed, the controller releases the bus and interrupts the processor.” The CGIF representation of these sentences is seen in Figure 5-13. The graphic representation of Figure 5-13 is seen in Figure 5-14.

```
[Proposition:''
[device:*a'dmac_DMA controller']
[value:*b'data']
[device:*c'memory_mem']
[action:*d'grant']
[device:*e'bus_systembus']
[action:*f'release']
[action:*g'interrupt']
[action:*h'request']
[action:*i'transfer']
[value:*j'command']
[device:*k'processor_proc']
[device:*l'iodev_peripheral i/o device']
[action:*m'send']
[state:*n''
[state:*o's_ready']
[device:?a'']
(mode?a?o)]
(operand?i?b)(agent?i?a)(enable_if?i?n)(destination?i?c)
(agent?h?a)(initiate_t?h?d)(patient?h?e)
(agent?d?k)(destination?d?a)(patient?d?e)(initiate_t?d?i)
(initiate_t?i?f)(initiate_t?i?g)
(patient?f?e)(agent?f?a)
(agent?g?a)(patient?g?k)
(contain?k?j)(initiate_t?m?h)(purpose?m?i)(operand?m?j)
(agent?m?k)(destination?m?l)]
```

Figure 5-13–Test graph from DMA patent in CGIF notation

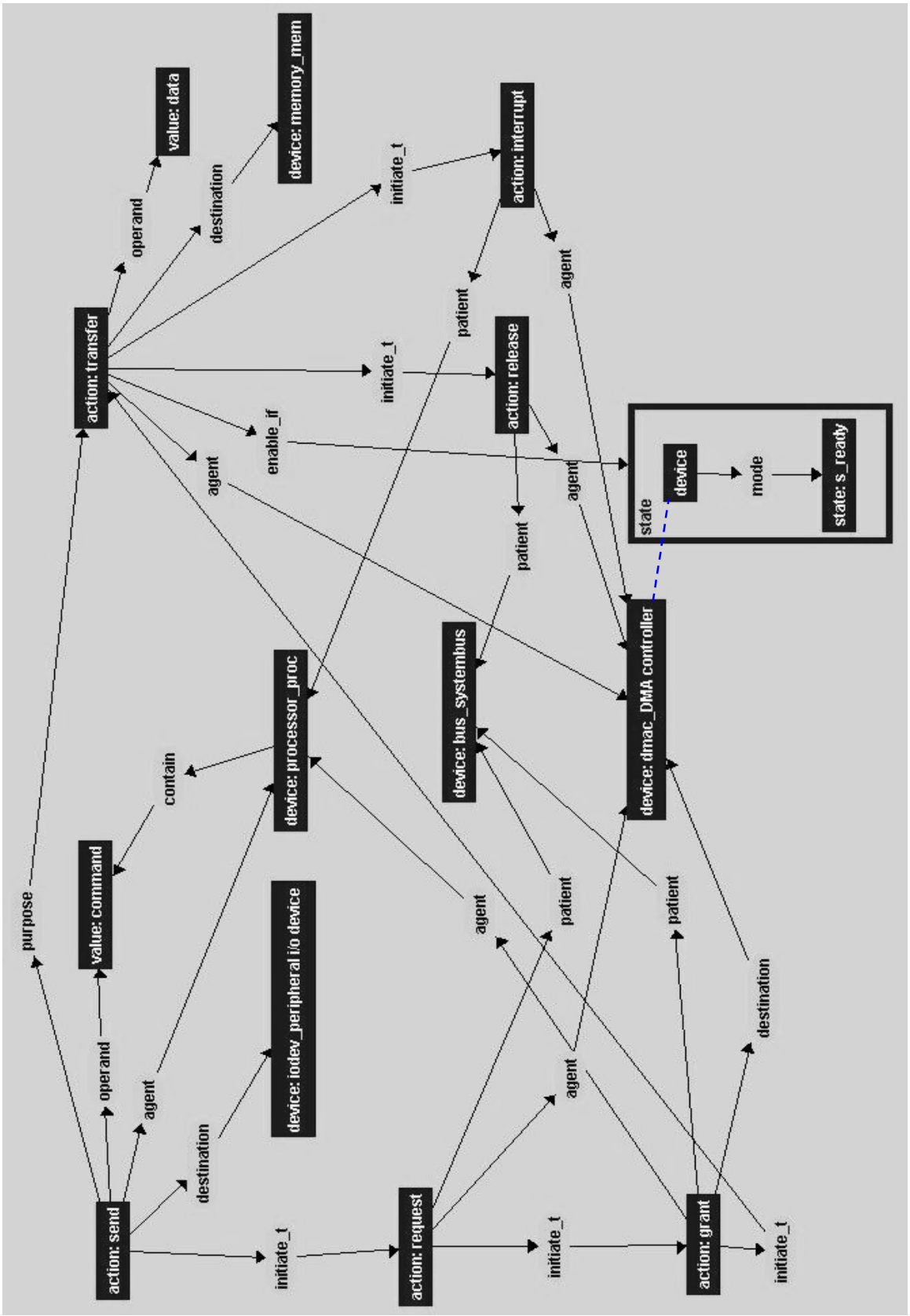

Figure 5-14–Test case from DMA patent in graphic form.

When the graph of Figure 5-14 was simulated, the simulator identified various errors in the graph. The steps taken to perform the simulation of this graph are described in this section. Also, this section will serve as a reference for future users of the simulator, as it considers the practical use of the simulator in simulating portions of technical documents.

As we can see from Figure 5-14, the conceptual graph involves device concepts such as DMA controller, i/o device and processor. Upon simulation of the graph, the simulator pointed out that the schema graphs for these device concepts were not present, and hence the simulation could not proceed. In response to these messages, schema graphs for these device concepts were created. When the simulator was supplied with schema graphs for all the device concepts, it was able to form a simulatable model for the conceptual graph.

In addition, the graph contains action words such as request, grant, release and interrupt. These action words did not have associated action procedures. When the graph was simulated, the simulator rightly pointed out that these action words did not have procedures associated with them, therefore simulation could not proceed. New action procedures were created for the words request, grant, release and interrupt. These action procedures were included in the simulator. The action word “send” also did not have an action procedure of its own. Since the task performed by the word “send” is the same as that of “transfer,” the procedure for the “transfer” action is called when the word “send” is encountered. When the simulator was provided with the above action procedures, the simulator got past the stage of identifying missing schemas and action procedures. It could begin the simulation.

At this stage, the simulator correctly identified a number of missing relations. For example, during the simulation of the “send” action, the processor acts as an agent for the action. Since a *source* relation is not present, the simulator assumes that agent for the “send” is also the source. This assumption is also logged to the output file, as a warning to the user. If the user feels that this assumption is not correct, he/she can edit the graph

and add a *source* relation for the “send” action. Similarly, the destination for the “send” is an i/o device, which has multiple values associated with it. The sent command can be stored in one of these multiple values. The information on which of these values is to be used is not present in the graph. Therefore, the simulator indicates to the user that a *target* relation is essential for simulation, without which simulation cannot proceed. The user now needs to edit the graph and add a value concept, which contains the address of the location to store the command. He also needs to add a new *target* relation, linking the value concept and the “send” action. After these modifications have been completed, the simulator is able to simulate the first sentence “First the processor sends a command to a peripheral i/o device to transfer data.”

The simulation of other sentences follows a similar pattern. The action words request, grant and release each require three essential relations, *agent*, *destination* and *patient.* Let us consider the “request” action to see why these three relations are necessary. In this case, the DMA controller is requesting the system bus from the processor. For the “request” action, the *agent* relation indicates the device that is performing the action of requesting, that is, the DMA controller. The *patient* relation is essential, as it indicates the device that is being requested. In this case, the system bus is being requested, and is the *patient* for this action. The *destination* relation is necessary, as it indicates the device from which the system bus is requested. The processor is the *destination* for the request action.

As we can see from Figure 5-14, the “request” action does not have an associated *destination* relation. The simulator is unable to proceed with the simulation, and it indicates that the “request” action has a missing *destination* relation. The user now needs to edit the graph and add the *destination* relation. After this, the second sentence “Next the DMA controller requests the bus” can be simulated. The effect of adding the *destination* relation is better understood by seeing its corresponding English sentence. With the addition of the *destination* relation to the graph, the graph models an improved second sentence that says, “Next the DMA controller requests the bus from the

processor.” The addition of the *destination* relation provides more information about the action, and makes it possible for the simulator to perform the action.

When all such missing information has been added to the graph, the simulator is able to simulate the graph completely. The modified graph is shown in CGIF notation in Figure 5-15, and in graphic form in Figure 5-16. The simulator output generated for this graph is shown in Figure 5-17.

```
[Proposition:''
[device:*a'dmac_DMA controller']
[action:*b'grant']
[device:*c'bus_systembus']
[action:*d'release']
[action:*e'interrupt']
[action:*f'request']
[action:*g'transfer']
[value:*h'command']
[device:*i'proc_processor']
[value:*j'data']
[device:*k'iodev_peripheral i/o device']
[action:*l'send']
[value:*m'v_5']
[device:*n'memory_mem']
[state:*o'']
[state:*p's_ready']
[device:?a'']
(eq?a?p)]
(contain?i?h)(purpose?l?g)(operand?l?h)(agent?l?i)(target?l?m)
(destination?l?k)(initiate_t?l?f)

(patient?f?c)(agent?f?a)(destination?f?i)(initiate_t?f?b)

(patient?b?c)(agent?b?i)(destination?b?a)(initiate_t?b?g)

(operand?g?j)(target?g?m)(location?g?m)(source?g?k)(agent?g?a)
(destination?g?n)(enable_if?g?o)(initiate_t?g?d)(initiate_t?g?e)

(destination?d?i)(patient?d?c)(agent?d?a)

(agent?e?a)(patient?e?i)]
```

Figure 5-15–Modified test case from DMA patent in CGIF notation.

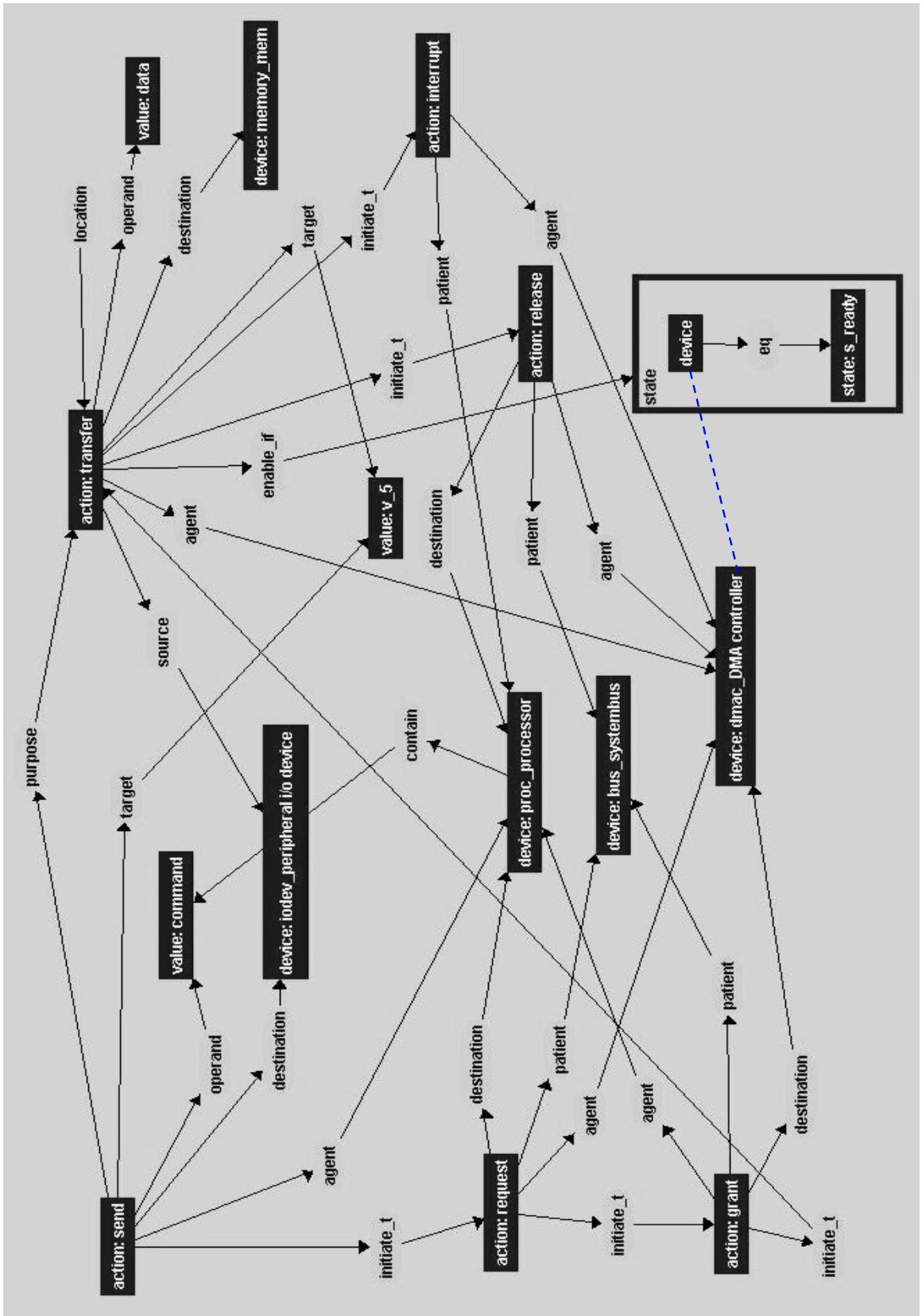

Figure 5-16–Modified test case from DMA patent in graphic form.

```
TIME: [0]
      STEP: [1]
            ACTION: send (mode = active)
      STEP: [2]
            DEVICE: iodev_peripheral i/o device (containAt = ‘0’)
            DEVICE: proc_processor (mode = busy)
      STEP: [3]
            DEVICE: proc_processor (mode = ready)
            ACTION: send (mode = inactive)
      STEP: [5]
            ACTION: request (mode = active)
      STEP: [6]
            DEVICE: dmac_DMA controller (busrequest = requested)
            DEVICE: proc_processor (busrequest = requested)
            DEVICE: dmac_DMA controller (mode = busy)
      STEP: [7]
            DEVICE: dmac_DMA controller (mode = ready)
            ACTION: request (mode = inactive)
      STEP: [9]
            ACTION: grant (mode = active)
      STEP: [10]
            DEVICE: proc_processor (busrequest = granted)
            DEVICE: dmac_DMA controller (busrequest = granted)
            DEVICE: bus_systembus (tristate = true)
            DEVICE: proc_processor (mode = busy)
      STEP: [11]
            DEVICE: proc_processor (mode = ready)
            ACTION: grant (mode = inactive)
      STEP: [13]
            ACTION: transfer (mode = active)
      STEP: [14]
            DEVICE: memory_mem (containAt = ‘0’)
            DEVICE: dmac_DMA controller (mode = busy)
      STEP: [15]
            DEVICE: dmac_DMA controller (mode = ready)
            ACTION: transfer (mode = inactive)
      STEP: [17]
            ACTION: release (mode = active)
            ACTION: interrupt (mode = active)
      STEP: [18]
            DEVICE: dmac_DMA controller(busrequest = notrequested)
            DEVICE: proc_processor (busrequest = notrequested)
            DEVICE: bus_systembus (tristate = true)
            DEVICE: dmac_DMA controller (mode = busy)
      STEP: [19]
            DEVICE: dmac_DMA controller (mode = ready)
            ACTION: release (mode = inactive)
FILE SIMULATION DONE!
```

Figure 5-17–Simulator output for the modified test case from DMA patent.

As seen from a comparison of Figures 5-14 and 5-16, the modified graph of Figure 5-16 contains all the missing concepts and relations that are needed to make the graph simulatable. Since no time delays have been specified in the graph, the entire simulation takes place at time 0. The different cycles of simulation are visible as different steps. We see that all the action words become “active” before execution and “inactive” after they complete their execution. Similarly, the agent concepts for these actions are “busy” during execution, and become “ready” after execution is completed.

From the above example test case chosen from the DMA patent, we can see that the simulator performed as expected. Through an iterative process of simulation and modification of the graph, the user can ensure that English descriptions of DMA systems can be accurately represented using conceptual graphs.

Chapter 6 Conclusions and Future Work

This chapter summarizes the advantages and disadvantages of the conceptual graph simulator. It looks at the present need for the simulator and the ways in which it can be incorporated into a design automation system. This chapter also makes some suggestions regarding future enhancements to the simulator.

6.1 Conclusions

The conceptual graph simulator described thus far is an important step in the validation of conceptual graph models. As stated earlier, the simulator was designed as a part of the design automation process that converts English language descriptions of digital systems to Hardware Description Language (HDL) code. This design automation process uses conceptual graphs as an intermediate stage of system representation. The simulator plays a significant role in three steps of the design automation process. These three steps are as follows.

First, the simulator is an effective tool for detecting semantic gaps in the English language description of systems. The ability with which a human reader can read and understand English is difficult to achieve using software. As an example, let us consider the pair of English sentences “The program counter contains the address of the next instruction to be fetched. The processor fetches the next instruction from memory.” A human reader experienced in computer systems almost instantaneously understands that the processor reads the next instruction from memory, using the contents of the program counter as the location to read from. However, natural language processing software is unable to make that link between the two sentences. Therefore, a conceptual graph based on the above sentences will not be simulatable, as the conceptual graph does not contain information about the location of the next instruction in memory. The simulator detects this lack of information and informs the user accordingly. The conceptual graph then

needs to be modified, so that the *fetch* knows that the location of the next instruction is stored in the program counter. The modified English sentences for this situation will be "The program counter contains the address of the next instruction to be fetched. The processor fetches the next instruction from the memory location stored in the program counter."

The simulator also plays a significant role is in the identification of logical errors. Suppose a digital system is described using a series of steps, and one of the steps contains a logical error, such as the data in a register being unintentionally over-written, or a memory location being wrongly cleared. When the description is converted to a conceptual graph, the graph also contains the logical error that is present in the description. When the user simulates the conceptual graph of the description, and examines the output generated by the simulator, he can find the logical error. This enables the user to correct the description before it can be converted to HDL code.

Finally, the simulator plays a role in closing the gap between the conceptual graph representation (of a system) and the HDL code. When considering HDL, we consider VHSIC Hardware Description Language (VHDL) in particular, as it is one of the commonly used HDLs, both by the academic and the industrial community. The role of the simulator in closing the above gap is illustrated by the following two points. First, the simulator treats each action word with a separate action procedure. This is consistent with the VHDL model of using different processes to perform different steps of the description. The concepts that control the execution of an action correspond to the items in the sensitivity list of a VHDL process. The concepts that enable the execution of an action correspond to the guard condition of a VHDL block. Second, the simulator relies on the existence of schema graphs for the simulation of device concepts. This is consistent with the VHDL method of using standard libraries to instantiate hardware components of a system. These two similarities greatly simplify the process of generating VHDL code from conceptual graphs.

The conceptual graph simulator described thus far depends mainly on two factors. These two factors are action procedures and device schema graphs. The behavior of a digital system may be described using a number of different actions. Each of these actions is vital to the system. Hence each of these actions need to have an associated action procedure. Thus, in order to simulate a conceptual graph of a complex system, we need a fairly large set of action procedures, describing about a hundred or so actions. This system is not yet equipped with that many action procedures. Second, a complex system consists of a variety of devices. Each of these devices plays a role in the behavior of the system. To simulate this complex system, the simulator also needs a fairly large library of device schema graphs. The library should be sufficient to describe the behavior of all the devices in the system. The present simulator is also limited in that aspect as it contains the schema graphs for only the frequently used register and memory concepts.

6.2 Future Enhancements

We have looked at the advantages and disadvantages of the present simulator. We now summarize the possible areas for future enhancements to the simulator. Enhancements to the simulator are suggested in three major areas. They are as follows.

First, in order to deal with graphs of complex and hierarchical systems, the simulator requires the presence of a comprehensive set of action procedures and device schema graphs. There are two possible methods for implementing this area of enhancement. The first involves a study of the present day digital systems. A future developer would study a range of digital systems, and then compile a list of devices and frequently occurring actions in these systems. Alternatively, the developer could obtain this information from readily available libraries, such as the VHDL libraries. By using the VHDL libraries, the task of the developer is significantly reduced, as he only needs to create schema graphs for the existing library components. Another advantage of using the VHDL libraries is that such device schema graphs are present in the VHDL libraries, thus simplifying the process of translation from conceptual graphs to VHDL. This list of

schema graphs and action procedures should be implemented in Java. This area of enhancement also involves changes to the simulator program so that it recognizes the presence of these new procedures and graphs and includes them during processing.

The second suggested area of enhancement is the development of a graphical user interface (GUI) to the simulator. As described in section 4.3, the present user interface is a command line interface. The command line interface presents two problems when dealing with the simulator. First, with a command line interface, the degree of interaction between the simulator is greatly reduced. In the presence of a GUI, the simulation may be made more interactive by allowing the user to "see" the simulation running. A GUI can be designed to allow the user to watch concepts during simulation, and keep track of their changes. A GUI can also allow the user a higher degree of control over the simulation, by providing breakpoints and the ability to step through the execution of actions.

The third possible area of enhancement would be to integrate this simulator with a conceptual graph editor. This would allow users to quickly implement changes in the conceptual graphs and run the simulation again, to ensure that the conceptual graph simulates correctly. As mentioned in section 5.2, the CharGer conceptual graph editor provides an easy-to-use interface for creating and editing conceptual graphs. This is one of the possible editors that the simulator could be integrated with. Any future development in this area will involve a study of the design of the editor. The major work in this area will center around the design of the interface between equivalent classes on the simulator side and the editor side.

This chapter concludes the document. We have examined the design and implementation of an object oriented conceptual graph simulator. The standards and software tools used for the implementation have been described. The simulator has been tested for its working, and the results of the experiments have been documented. The advantages and disadvantages of the simulator have been presented, and possible areas for future enhancements have been listed. The references and the appendices for the document follow this chapter.

References

Bos, C., Botella, B., and Vanheeghe, P. (1997). Modeling and Simulating Human Behaviours with Conceptual Graphs. *Proceedings of the Fifth International Conference on Conceptual Structures* (pp. 275-289). Berlin: Springer-Verlag.

Cyre, W. R., Balachandar, S., & Thakar, A. (1994). Knowledge Visualization from Conceptual Structures. *Proceedings of the Second International Conference on Conceptual Structures* (pp. 275-292). Berlin: Springer-Verlag.

Cyre, W. R. (1998). Executing Conceptual Graphs. *Proceedings of the Sixth International Conference on Conceptual Structures* (pp. 51-64). Berlin: Springer-Verlag.

Delugach, H. S. (1990). Using Conceptual Graphs to Analyze Multiple Views of Software Requirements. Proceedings of the Sixth Workshop on Conceptual Structures, Boston, MA: Springer Verlag.

Delugach, H. S. (1991). Dynamic Assertion and Retraction of Conceptual Graphs. *Proceedings of the Seventh Workshop on Conceptual Structures*. Binghamton, NY: Springer Verlag.

Delugach, H. S. (2000). http://www.cs.uah.edu/~delugach/CharGer/ *The CharGer conceptual graph editor*. Date: 09/07/2000

Harel, D., and Naamad, A. (1995). *The STATEMATE Semantics of Statecharts*. Andover, MA: i-Logix, Inc.

Kamath, R., and Cyre, W. R. (1995). Automatic Integration of Digital Systems Requirements using Schemata. *Proceedings of the Third International Conference on Conceptual Structures* (pp. 44-58). Berlin: Springer-Verlag.

Law, A. M., and Kelton, W. D. (1991). *Simulation Modeling and Analysis*, 2nd edition. McGraw Hill, Inc.

Lipsett, R., Schaefer, C. F., and Ussery, C. (1989). *VHDL: Hardware Description and Design.* Boston, MA: Kluwer Academic.

Lukose, D. (1993). Executable Conceptual Structures. *Proceedings of the First International Conference on Conceptual Structures* (pp. 223-237). Berlin: Springer-Verlag.

Raban, R., and Delugach, H. S. (1997). Animating Conceptual Graphs. *Proceedings of the Fifth International Conference on Conceptual Structures* (pp. 431-445). Berlin: Springer-Verlag.

Rational Software Corp. (2001). http://www.rational.com/products/rose/ Rational Rose Visual Modeling Tool.

Schmidt, J. W., and Taylor, R. E. (1970). *Simulation and Analysis of Industrial Systems*, Richard D. Irwin, Homewood, IL.

Schriber, T. J. and Brunner, D. T. (1996). Inside Simulation Software: How it Works and Why it Matters. *Proceedings of the 1996 Winter Simulation Conference.* Society for Computer Simulation.

Southey, F. and Linders, J. G. (1999). NOTIO–A Java API for Developing CG Tools. *Proceedings of the Seventh International Conference on Conceptual Structures* (pp. 262-271). Blacksburg, VA: Springer Verlag.

Southey, F. (2000). http://backtrack.math.uwaterloo.ca/CG/projects/notio *Notio Project Page.* Date: 06/06/2000

Sowa, J. F. (1984). *Conceptual Structures.* Reading, MA: Addison-Wesley.

Sowa, J. F. (1999). Conceptual Graphs: Draft Proposed American National Standard. *Proceedings of the Seventh International Conference on Conceptual Structures* (pp. 1-65). Blacksburg, VA: Springer Verlag

Sowa, J. F. (2001). http://www.bestweb.net/~sowa/cg/cgstand.htm *A Working Draft of the Proposed ISO Conceptual Graph Standard.* Date: 04/06/2001

Sowa, J. F. (2001). http://www.bestweb.net/~sowa/cg/tut.htm *Tutorial on Conceptual Graphs.* Date: 04/05/2001

Appendix A – Notes to Users

This appendix contains notes to the user on how to use the simulator. It lists the various modes of operation available, and how the user can use these modes.

The simulator is available as a compressed (zip) file containing all the source (java) files and three batch files: sim.bat, comp.bat, and run.bat. The compressed file has been created using Winzip™ version 7.0. All these files need to be unzipped into a single folder. The computer needs to have JDK 1.3 or higher installed on it to compile and run the application. The conceptual graph to be tested must be represented in CGIF. The file that stores the graph must be in the same folder as above.

The preparatory steps that the user must take are as follows.

1) Open a DOS window and navigate to the above folder
2) At the prompt, type “path” and hit the Enter key.
3) The path should contain entries to the folder where JDK is installed.
4) At the prompt, type “comp” and hit the Enter key. This should compile all the files.

Once these steps have been successfully completed, the software is ready to begin simulation. Let us say the graph is stored in a file with the name “graph.cgf.” The software may be run in two possible modes, normal or debug. The default mode is normal. Also, the simulator may be configured for three different output-options, less, medium, or more. The default output-option is less. The command to type for a given mode and output-option is given in Table A-1. Once the command has been determined from Table A-1, the user must perform the following steps.

1) At the prompt of the same folder, type the command and hit the Enter key
2) The simulation proceeds, and the results are logged to the file “cgsimlog.txt”

Note that the output is always logged to the file “cgsimlog.txt.” It is advisable to save this file under another name after the completion of simulation, so that the simulation results are not lost when the simulator is run again.

Table A-1–Commands to be used.

(For the different modes and output options of the simulator.)

Number	Mode of simulator	Output option chosen	Command to type
1.	normal	default (less)	run graph.cgf
2.	normal	less	run graph.cgf less
3.	normal	medium	run graph.cgf medium
4.	normal	more	run graph.cgf more
5.	debug	default(less)	run graph.cgf –d
6.	debug	less	run graph.cgf –d less
7.	debug	medium	run graph.cgf –d medium
8.	debug	more	run graph.cgf –d more

Appendix B – Notes to Future Developers

This appendix contains design details for future developers. First, the algorithm for the simulation engine is presented. Naming conventions for the different concepts are considered next. Design steps for device schema graphs form the third area of interest. Finally, areas of code that need changes to include more action procedures and device schema graphs are identified.

Algorithm for the Simulation Engine

```
01. Prepare the graph
02. Copy the initial incidents to the incident queue
03. While not end of simulation do the following
   a) Move incidents at current simulation time to current queue
   b) Process current incidents, which may generate new incidents
   c) Add the newly generated incidents to the incident queue

**************************************************************
```

Preparing the graph:

```
01. Process all device concepts in the graph. For each device
    concept, perform a join with its corresponding schema
02. If concept is a dependent concept, do nothing
03. If concept is a context, prepare its descriptor graph
04. If concept is a value concept, prepare the value concept
05. If concept is a state concept, prepare the state concept
06. If concept is an action concept, prepare the action concept
07. If concept is an event concept, do nothing
```

Preparing the value concept:

```
01. If the referent begins with a "c_", this is a value concept
    added during device preparation. Store the value after the
    "c_" in the concept and end
```

02. This is a value concept present in the graph before device preparation. If the concept begins with a “v_”, its value is the value after the “v_”. Else assign a random value to it
03. Create a new ValueConcept with this value
04. Add an "eval" relation to the value concept, linking it to the new ValueConcept
05. Add the “eval” relation to the graph

Preparing the state concept:

01. If the referent begins with a “c_”, this is a state concept added during device preparation. Store the string after the “c_” in the concept and end
02. This is a state concept present in the graph before device preparation. If the concept begins with a “s_”, its state is the string after the “s_”. Else assign “undefined” state to it
03. Create a new StateConcept with this state
04. Add an "eval" relation to the state concept, linking it to the new StateConcept
05. Add the “eval” relation to the graph

Preparing the action concept:

01. Create a new StateConcept with “inactive” state
02. Add an "mode" relation to the action concept, linking it to the new StateConcept
03. Add the “eval” relation to the graph

End Of Simulation Determination:

End of simulation is reached when either of the following conditions becomes true:

01. The incident queue is empty

02. The simulation time crosses a prefixed maximum time

Current Simulation Time Determination:

It is determined as the activation time of the first incident of the incident queue.

Processing Current Incidents:

01. Process value incidents
02. Process state incidents
03. Process event incidents
04. Process action incidents
05. Clear the Fired Event List

Processing Value Incidents:

For each value incident in the current queue, do the following.

01. Read the identifier and identify the concept it refers to
02. If the concept is not a dominant concept, get its dominant concept
03. Change the value of the ValueConcept to the value of the ValueConcept in the value incident

Processing State Incidents:

For each state incident in the current queue, do the following.

01. Read the identifier and identify the concept it refers to
02. If the concept is not a dominant concept, get its dominant concept
03. Change the state of the StateConcept to the state of the StateConcept in the state incident
04. If an action initiation state incident, trigger all outgoing relations with no modifier and call the action procedure

05. If an action suspension state incident, trigger all outgoing relations with modifier “_s”
06. If an action resumption state incident, trigger all outgoing relations with modifier “_r”
07. If an action completion state incident, trigger all outgoing relations with modifier “_t”

Processing Event Incidents:

For each event incident in the current queue, do the following

01. Read the identifier and identify the concept it refers to
02. If the concept is not a dominant concept, get its dominant concept
03. Add the event to a list of fired events
04. Fire the event concept with a queue for any generated incidents

Processing Action Incidents:

For each action incident in the current queue, do the following

01. Read the identifier and identify the concept it refers to
02. If the concept is not dominant, get its dominant concept
03. Determine the type of incident
    a) If initiate incident, create an action initiation state incident
    b) If suspend incident, create an action suspension state incident, and move all its incidents to Suspended Queue
    c) If resume incident, create an action resumption state incident, move all its incidents back to Incident Queue
    d) If terminate incident, create an action completion state incident, remove all incidents from all queues.

Calling the action procedure:

01. Get the name of the action
02. Find the corresponding procedure in the set of available procedures

```
03. Call the corresponding procedure
```

Firing an event concept:

```
01. Create modified relations: append modifier to base relations
02. For each outgoing relation do the following
     a) If the relation is the modified initiate relation, add an
        incident to initiate an action in the next step
     b) If the relation is the modified suspend relation, add an
        incident to suspend an action in the next step
     c) If the relation is the modified resume relation, add an
        incident to resume an action in the next step
     d) If the relation is the modified terminate relation, add an
        incident to stop an action in the next step
     e) If the relation is the modified trigger relation, add an
        incident to trigger an event in the next step
```

Naming Conventions for Concepts

1. For value concepts
   a) If they are objects of class ValueConcepts, and the developer wants to ensure that the concept starts simulation with a known value, such a concept should be named with a “c_” followed by the value that has to be stored in the concept
   b) If they are not objects of class ValueConcepts, and the developer wants to ensure that the concept has a known value at the start of simulation, such a concept should be named with a “v_” followed by the value that has to be stored in it
   c) Other value concepts get a value from a set of 10 random values

2. For state concepts
   a) If they are objects of class StateConcepts, and the developer wants to ensure that the concept starts simulation in a known state, such a concept should be named with a “c_” followed by the state that has to be stored in the concept

b) If they are not objects of class StateConcepts, and the developer wants to ensure that the concept has a known state at the start of simulation, such a concept should be named with a "s_" followed by the state that has to be stored in it
c) Other state concepts get a state that is "undefined"

3. There are no naming conventions for event and action concepts

4. For device concepts
   a) all device names that begin with "register_" are joined to a schema for registers
   b) all device names that begin with "memory_" are joined to a schema for memory
   c) all device names that begin with "bus_" are joined to a schema for bus

---

Design Steps for Device Schema Graphs

In general, the device schema graphs use the specialized concepts for device, value, and state concepts. This is done for the reason that the once the device has been expanded by its schema graph, there is no need to add additional concepts for simulating the device. Hence all value concepts in a device use the ValueConcept class instead of just the concept class. All state concepts in a device use the StateConcept class. The head concept of a schema graph is a DeviceConcept.

The Register schema graph consists of a DeviceConcept and a ValueConcept, linked by a *contain* relation. The device concept has no particular referent, but the value concept in the Register has a referent of "c_" followed by the initial value it should hold. The constructors for the class support either loading a default value or a user-supplied value as the initial value.

The Memory schema graph consists of a DeviceConcept and a number of ValueConcepts pairs (for the value and its index). The *containAt* relation relates these ValueConcept pairs to the DeviceConcept. All ValueConcepts in this graph have a referent of "c_" followed by the initial value. The *size* relation links another

ValueConcept that holds the size of the memory to the DeviceConcept. The default number of elements in memory is set to sixteen. The constructors for the class allow for the user to set the size of memory and choose the initial contents of the memory.

The Bus schema graph consists of a DeviceConcept and a ValueConcept, linked by a *contain* relation. The device concept has no particular referent, but the value concept in the Register has a referent of "c_" followed by the initial value it should hold. The constructors for the class support either loading a default value or a user-supplied value as the initial value. The DeviceConcept is also linked to a StateConcept, using a *tristate* relation. The StateConcept has a referent of "c_true", meaning that the bus is initially tristated. This will be useful when the notion of requesting and granting busses is used.

When designing future schema graphs, the developer needs to note that maximum freedom is given to the person who creates the main conceptual graph (the graph being simulated). Hence the developer should try to create as many constructors as possible to allow the user a wide variety of options in integrating these schema graphs.

All schema graphs for device MUST contain a method called GetMainConcept(). This method returns the head concept of the schema graph. This is needed to ensure uniform handling for existing and future device concepts.

Code Areas to be updated to add Action Procedures and Schema Graphs

The code area to be updated when more action procedures are added is in the SimulationEngine.java source file. The name of the method is ExecuteAction(). The developer needs to add an "else if" construct and append the new action procedure.

The code area to be updated when more device schema graphs are added is also in the SimulationEngine.java source file. The name of the method is prepare(). This method first prepares all device concepts. The developer needs to add an "else if" construct and then include the code for joining the new schema graph for the device. Also, the

developer needs to use a standard string (such as “register_” for registers, “memory_” for memory) that the code matches for, and document the change, so that future developers may not use the same term again.

Appendix C – Pseudo-code for Selected Action Procedures

This appendix contains algorithms for the action procedures of commonly occurring actions. The words are transfer, read, write, execute, increment, clear, request, and grant. The phrase "status of the action" is used to indicate successful simulation of the action.

Transfer action

A transfer deals with 1 or multiple values being transferred. The steps performed for a transfer action are as follows.

1. see if an *if* relation exists, if not, move to step 2
   a) check the veracity of the enabling condition
   b) if it is false, move to step 10
   c) if it is true, move to step 2
2. see if a *delay* relation exists, if so, store delay time for future use
3. see if a *quant* relation exists, if so, store the number of values to be transferred
4. find the *source* relation, if not found, return the status of the action as "error"
   a) if quant > 1 && the source is a value or a register, return the status of the action as "error"
   b) if quant > 1 && the source is memory, check for *location* relation, if *location* relation does not exist, return the status of the action as "error"
5. find the *destination* relation, if not found, return the status of the action as "error"
   a) if quant > 1 && the destination is a value or a register, return the status of the action as "error"
   b) if quant > 1 && the destination is memory, check for *target* relation, if *target* relation does not exist, return the status of the action as "error"
6. see if an *instrument* relation exists, if so, store the fact that the *instrument* relation exists
7. create value incidents for all the transfers to be performed
8. if an *instrument* relation exists, create value incidents for the carrier also

9. create state incidents for action initiation and completion
10. return the status of the action of the action as “success”

Read action

A read deals with 1 value being read. The same action procedure can be called for action words like fetch, get, pull, pop, latch, and take. The steps performed for a read action are similar to the steps for a transfer action, except that we need not check for multiple values.

Also, for a read action, if there is no destination, we assume the agent for the action stores the value that is read. Hence, the presence of either a destination or an agent relation is essential. The steps for performing a read action are as follows.

1. see if an *if* relation exists, if not, move to step 2
    a) check the veracity of the enabling condition
    b) if it is false, move to step 10
    c) if it is true, move to step 2
2. see if a *delay* relation exists, if so, store delay time for future use
3. find the *source* relation, if not found, return the status of the action as “error”
    a) if the source is memory, check for *location* relation, if *location* relation does not exist, return the status of the action as “error”
4. find the *destination* relation, if not found, move to step 5
    a) if the destination is memory, check for *target* relation, if *target* relation does not exist, return the status of the action as “error”
5. find the *agent* relation, if not found, and *destination* relation not found, return the status of the action as “error”, else move to step 6
    a) check if a *contain* relation is associated with the agent, if not, return the status of the action of the action as “error”
6. see if an *instrument* relation exists, if so, store the fact that the *instrument* relation exists
7. create a value incident for the value to be read

8. if an *instrument* relation exists, create a value incident for the carrier also
9. create state incidents for action initiation and completion
10. return the status of the action as “success”

Write action

A write deals with one value being written. The same action procedure can be called for action words like store, put, push, and give. The steps performed for a write action are similar to the steps for a transfer action, except that we need not check for multiple values.

Also, for a write action, if there is no source, we assume the agent for the action stores the value that is written. Hence, the presence of either a source or an agent relation is essential. The steps for performing a write action are as follows.

1. see if an *if* relation exists, if not, move to step 2
   a) check the veracity of the enabling condition
   b) if it is false, move to step 10
   c) if it is true, move to step 2
2. see if a *delay* relation exists, if so, store delay time for future use
3. find the *source* relation, if not found, move to step 4
   a) if the source is memory, check for *location* relation, if *location* relation does not exist, return the status of the action as “error”
4. find the *agent* relation, if not found, and *source* relation not found, return the status of the action as “error”, else move to step 5
   a) check if a *contain* relation is associated with the agent, if not, return the status of the action of the action as “error”
5. find the *destination* relation, if not found, return the status of the action as “error”
   a) if the destination is memory, check for *target* relation, if *target* relation does not exist, return the status of the action as “error”
6. see if an *instrument* relation exists, if so, store the fact that the *instrument* relation exists

7. create a value incident for the value to be written
8. if an *instrument* relation exists, create a value incident for the carrier also
9. create state incidents for action initiation and completion
10. return the status of the action as “success”

Execute action

An execute action is used to perform the execution of simple arithmetic, Boolean, and logical commands. The command to be performed is specified using the *operation* relation. The arithmetic commands performed are addition, subtraction, multiplication, division, modulus, and exponentiation. The Boolean commands perform the logical AND and the logical OR of two state operands.

The logical commands are of 2 types. The numeric logical commands perform comparisons like greater than (>), lesser than (<), greater than or equal to (>=), lesser than or equal to (<=), equal to (==) and not equal to (!=). The string logical commands compare the equality or inequality of two string operands. The string logical commands may be used to compare state concepts. All the logical commands result in Boolean outputs, which may be stored either in a state or a value variable. The steps for performing are listed below.

1. see if an *if* relation exists, if not, move to step 2
    d) check the veracity of the enabling condition
    e) if it is false, move to step 11
    f) if it is true, move to step 2
2. see if a *delay* relation exists, if so, store delay time for future use
3. find the *operation* relation, if not found, return the status of the action as “error”
    a) determine whether the operation is arithmetic or logical
    b) if logical, determine whether the operation is arithmetic, Boolean or string logical
4. find the *operand1* relation, if not found, return the status of the action as “error”
    a) if operation and operand1 type disagree, return the status of the action as “error”
5. find the *operand2* relation, if not found, return the status of the action as “error”

   a) if operation and operand2 type disagree, return the status of the action as “error”
6. see is a *result* relation exists, if not, warn the user about the missing *result* relation
7. perform the operation on the 2 operands
8. if a result exists, create an incident to store the result in the result concept
9. if an *instrument* relation exists, create a value incident for the carrier also
10. create state incidents for action initiation and completion
11. return the status of the action as “success”

Increment action

An increment action is used to perform the incrementing of a value operand. The value operand may be part of value concepts or device concepts. If the value to be incremented is contained in a device concept such as memory, we also need to verify the existence of a location relation to pick out the correct operand from memory. The steps for performing an increment operation are listed below.

1. see if an *if* relation exists, if not, move to step 2
   a) check the veracity of the enabling condition
   b) if it is false, move to step 10
   c) if it is true, move to step 2
2. see if a *delay* relation exists, if so, store delay time for future use
3. find the *operand* relation, if not found, return the status of the action as “error”
   a) if the first operand points to memory, search for a *location* relation
   b) if the relation is not found, return the status of the action as “error”
4. see is a *result* relation exists, if not, warn the user about the missing *result* relation
   a) if it exists, and it points to memory, search for a *target* relation
   b) if the *target* relation is not found, return the status of the action as “error”
5. increment the operand
6. if a result exists, create an incident to store the result in the result concept
7. if an *instrument* relation exists, create a value incident for the carrier also
8. create state incidents for action initiation and completion
9. return the status of the action as “success”

Clear action

A clear action is a specialized form of the write action, as it deals with setting a given destination to zero. The same action procedure can be called for action words like initialize and reset. The steps performed for a clear action are similar to the steps for a write action, except that we need not check for a source relation. The steps for performing a write action are as follows.

1. see if an *if* relation exists, if not, move to step 2
   a) check the veracity of the enabling condition
   b) if it is false, move to step 8
   c) if it is true, move to step 2
2. see if a *delay* relation exists, if so, store delay time for future use
3. find the *operand* relation, if not found, return the status of the action as "error"
   a) if the operand is memory, check for *target* relation
   b) if *target* relation does not exist, return the status of the action as "error"
4. see if an *instrument* relation exists, if so, store the fact that the *instrument* relation exists
5. create a value incident for the value to be cleared
6. if an *instrument* relation exists, create a value incident for the carrier also
7. create state incidents for action initiation and completion
8. return the status of the action as "success"

Request action

A request action is used by devices such as DMA controllers to request the bus from the processor. When the request action is executed, the action checks for the presence of the *agent*, *destination* and *patient* relations. All of these are essential relations. The *patient* relation specifies the device that is being requested, the *agent* relation specifies the device that is requesting the *patient* device, and the *destination*

relation specifies the device that has current control of the *patient* device. The steps for performing a write action are as follows.

1. see if an *if* relation exists, if not, move to step 2
    a) check the veracity of the enabling condition
    b) if it is false, move to step 8
    c) if it is true, move to step 2
2. see if a *delay* relation exists, if so, store delay time for future use
3. find the *agent* relation, if not found, return the status of the action as “error”
4. find the *destination* relation, if not found, return the status of the action as “error”
5. find the *patient* relation, if not found, return the status of the action as “error”
6. create state incidents to change the *busrequest* status of the agent and the destination to “requested”
7. create state incidents for action initiation and completion
8. return the status of the action as “success”

Grant action

A grant action is used by devices such as processors to grant the bus to requesting devices such as DMA controllers. When the grant action is executed, the action checks for the presence of the *agent*, *destination* and *patient* relations. All of these are essential relations, as described in the request action. The steps for performing a write action are as follows.

1. see if an *if* relation exists, if not, move to step 2
    a) check the veracity of the enabling condition
    b) if it is false, move to step 9
    c) if it is true, move to step 2
2. see if a *delay* relation exists, if so, store delay time for future use
3. find the *agent* relation, if not found, return the status of the action as “error”
4. find the *destination* relation, if not found, return the status of the action as “error”

5. find the *patient* relation, if not found, return the status of the action as “error”
6. create state incidents to change the *busrequest* status of the agent and the destination to “granted”
7. create state incidents to change the *tristate* concept of the patient to “true”, meaning the bus (the patient) has been tristated before granting it
8. create state incidents for action initiation and completion
9. return the status of the action as “success”

Appendix D – Listing of the UML Documentation

This appendix lists the UML-generated documentation for the various classes of the conceptual graph simulator.

Selected Logical View Report

Logical View

CGSimulator

This is the main class that contains the conceptual graph simulator. The class reads the graph from a file (using a FileInput class), simulates it (using the SimulationEngine class) and logs all the output of the simulation to a file (using the FileOutput class)

Public Methods:

main (args : String[] = default) : void

This is the main function in the class.
It accepts one command line argument, and returns nothing

Arguments: 1
1. String[] args:
args[0] holds the filename of the file to be simulated
args[1] if present may hold one of the following
"-d" implies debug mode (steps through simulation)
"less" implies less output
"medium" implies moderate output
"more" implies lots of output
args [2] if present may hold one of the following
"less" implies less output
"medium" implies moderate output
"more" implies lots of output

```
Returns: nothing
```

Selected Logical View Report

Logical View

FileInput

This class contains the functions for reading a conceptual graph from the supplied file, and returning it.

Public Methods:

getGraph (fileName : String) : notio.Graph

This function opens the supplied file and reads a conceptual graph (in CGIF) from it, and returns the graph using Notio classes.

Arguments: 1

1. String fileName: the name of the file containing the graph

Returns: a Notio Graph object corresponding to the graph in the file

Selected Logical View Report

Logical View

FileOutput

This class is used for logging the results of simulation to a file. The class uses two static methods, which may be called from anywhere.

Public Methods:

openLogFile () : void

The openLogFile method is used to open a log file and initialize it. The log file is always opened with the name "cgsimlog.txt".

After opening the file, the method records the current date and time into the file.

Arguments: 0

Returns: nothing

log (data : String = default) : void

This method is used to log a String of data into the file. It accepts a string, and then appends to the string to the file and then appends a new line to the file.

Arguments: 1

1. data: this is a string argument representing the string of data that should be logged to the file

Returns: nothing

Selected Logical View Report

Logical View

SimulationEngine

This class is the mos important class of the simulator. It contains the actual simulation engine, and performs the simulation using queues.

The constructor of this class accepts the graph to be simulated, the output option, and the debug flag. It then creates space for all queues, prepares the graph, and sets all its internal output options based on the external output option, and also sets its mode to normal or debug, based on the external debug flag.

Private Properties:

simtime, simstep : int = 0

These two variables represent the current simulation time and simulation step. The two variables are initialized to zero.

cg : notio.Graph

This variable holds the graph to be simulated.

correctlyExecuted : boolean = true

This boolean is used to ensure that each and every stage of the simulation executes correctly. If this variable goes false, all the queues are cleared, and simulation stops.

CurrentQueue : Vector

This queue is used to hold the incidents that have to be processed at the current simulation time.

IncidentQueue, NextQueue : Vector

These queues are used to hold incidents. The IncidentQueue is the main queue, it holds all the incidents. The NextQueue is used to hold all newly generated incidents within one simulation cycle. After the cycle completes, all incidents in the NextQueue are merged into the IncidentQueue.

SuspendedQueue, FiredEventQueue : Vector

The SuspendedQueue is used to store the incidents of all the suspended actions. The FiredEventQueue stores all the events that were fired during the current simulation cycle.

ms : notio.MatchingScheme

This MatchingScheme is used when performing graph joins, when device concepts have to be prepared by expanding them with their schema graphs. This helps in matching concepts and relations based on the values it contains.

cs : notio.CopyingScheme

This CopyingScheme is used when performing graph joins, when device concepts have to be prepared by expanding them with their schema graphs. This determines how events are copied from the joining graphs to the resulting graph.

less, medium, more, debug : Boolean = false

These booleans hold values that make the simulator more user-friendly. The less variable, if true,

instructs the simulator to log less output to the log file. The medium and more variables, if true, log a medium amount of, and more amount of output, to the log file.

The debug variable, if true, allows the user to step through the simulation by periodically requesting user input from the keyboard to continue simulation.

MaxTime, MaxStep : int = 10000
They represent the maximum time and step that the simulation will proceed for, before the simulator ends. These variables are set to 10000, and cannot be changed. They are necessary if the simulator goes into an infinite loop.

cta, ctd, cte, cts, ctv : notio.ConceptType
These variables represent the concept types for action, device, event, state, and value concepts respectively.

cth : notio.ConceptTypeHierarchy
This variable stores the concept type hierarchy for the simulator. At present, it just has the 5 basic concept types added to it, apart from the universal and absurd types.

EvalRelationType : notio.RelationType
This variable holds the relation type for creating a new "eval" relation. This is used when preparing value and state concepts.

ModeRelationType : notio.RelationType

This variable holds the relation type for creating a new "mode" relation. This is used when preparing action concepts.

Public Methods:

simulate (InitIncidents : Vector) : void

This method is called to simulate the graph. A set of initial incidents is supplied to the method.

Arguments: 1

1. InitIncidents: the set of initial incidents

Returns: nothing

Private Methods:

UpdateQueues (phase : int) : void

This method is used to update the three basic simulator queues. This updating is done in 2 phases. This method takes one parameter that indicates the phase to perform. Phase 1 selects incidents from the IncidentQueue at the current simulation time and loads them into the CurrentQueue. Phase 2 takes place after the current incidents have been processed. This phase merges incidents from the NextQueue to the IncidentQueue.

Arguments: 1

1. phase: indicates which phase to perform

Returns: nothing

ProcessCurrentIncidents () : void

This method processes the current incidents. It takes one parameter, the queue name. It then calls four different operations to process value, state, event, and action incidents.

Arguments: 1
1. queue: the queue of incidents to be processed

Returns: nothing.

ProcessValueIncidents () : void
This method processes the current value incidents. It takes one parameter, the queue name. From the queue, it removes all the value incidents and processes them.

Arguments: 1
1. queue: the queue of incidents to be processed

Returns: nothing.

ProcessStateIncidents () : void
This method processes the current state incidents. It takes one parameter, the queue name. From the queue, it removes all the state incidents and processes them.

If the state incident represents the activity of an action, then the simulator behaves as if an event has occurred, and calls the fire operation for the action concept. Additionally, if an action has been initiated, it calls the ExecuteAction for the concept.

Arguments: 1
1. queue: the queue of incidents to be processed

Returns: nothing.

ProcessEventIncidents () : void

This method processes the current event incidents. It takes one parameter, the queue name. From the queue, it removes all the event incidents and processes them.

Arguments: 1

1. queue: the queue of incidents to be processed

Returns: nothing.

ProcessActionIncidents () : void

This method processes the current action incidents. It takes one parameter, the queue name. From the queue, it removes all the action incidents and processes them.

Depending on the action type, action are either initiated, suspended, resumed or terminated. Each of these are processed by creating the correct state incident for them.

Arguments: 1

1. queue: the queue of incidents to be processed

Returns: nothing.

ExecuteAction (c : notio.Concept) : void

This method is used to check for the action name and then call the relevant action procedure from a list of action procedures.

Arguments: 1

1. c: the action concept to execute

Returns: nothing

fire (ac : notio.Concept, time : int, step : int, queue : Vector, modifier : String) : boolean
This method is called to check all outgoing propagational relations, and generate new incidents. This method is used to fire event concepts. Also, activity changes for action concepts are also treated as event concepts and this method is called.

Arguments: 5
1. ac: the concept whose outgoing relations have to be checked
2. time: the current simulation time
3. step: the current simulation step
4. queue: the queue to which new incidents have to be added
5. modifier: the relation modifier to take care of activity changes of action concepts

Returns: a boolean value, saying whether the fire was successful or not

evaluate (c : notio.Concept) : boolean
This method is called to evaluate the veracity of state concepts. This is mainly used to perform nested simulation and verify the veracity of nested graphs.

Arguments: 1
1. c: the state concept whose veracity is needed

Returns: a boolean saying whether evaluation completed successfully or not

prepare (cg : notio.Graph) : notio.Graph
This method is called to prepare a conceptual graph for simulation. The method adds new concepts for preparing value, state and action concepts. It prepares device concepts by joining them with their schema graph.

Arguments: 1
1. cg: the graph to be prepared

Returns: the prepared graph.

testCoreference (c : notio.Concept) : notio.Concept
This method is called to test the coreference of a concept. This is used both during preparing the graph and simulating it. This method returns the defining concept of the coreference set that the input concept belongs to. If it does not belong to any coreference set, it returns the concept itself.
Arguments: 1
1. c: the concept whose coreference has to be tested

Returns: the defining concept of c, or c itself.

Selected Logical View Report

Logical View

Incident

This class concerns the representation of incidents.

The class has six attributes and eight member functions.

These functions mainly deal with accessing the attributes of the class.

Private Properties:

activationTime : int

This represents the time at which the incident should be processed.

step : int

This represents the step at which the incident should be processed.

level : int

This holds the level of nesting of the incident. What this implies is whether the incident belongs to the outermost graph, or whether it belongs to an inner (nested) graph.

type : String

This attribute of the incident stores the type of concept the incident refers to. The different types of cncepts that represent valid values are "value", "state", "event" and "action."

object : Object

This attribute stores the actual concept in the graph that the incident represents.

concept : Object

This attribute stores the concept in the graph that is responsible for the generation of the incident. This is particularly useful when suspending, resuming and terminating actions.

Public Methods:

getActivationTime () : int

This method returns the activation time of the incident.

Arguments: 0

Returns: time at which the incident is processed

getStep () : int

This method returns the step at which the incident is processed.

Arguments: 0

Returns: the step at which an incident is processed

getLevel () : int

This method returns the level of the incident, that is, whether the incident represents an inner (nested) graph or the outer graph.

Arguments: 0

Returns: the level of nesting on the incident

getObjectType () : String
This method returns the type of the incident, whether it is a value, state, event or action incident.

Arguments: 0

Returns: a string representing the type of incident.

getObject () : Object
This method returns the object (concept) that this incident is supposed to represent.

Arguments: 0

Returns: an Object representing the concept

getConcept () : Object
This method returns the concept that is responsible for generating the incident.

Arguments: 0

Returns: the concept that generated the incident

setActivationStep (newTime : int) : void
This method is used to set the activation time of the incident. The main use of this method is during the processing of the suspend action incident.

Arguments: 1
1. newActivationTime: the new time at which the incident has to be activated

Returns: nothing

setStep (newStep : int) : void

This method is used to set the step of the incident. Mainly used during the processing of incidents in the SuspendedQueue.

Arguments: 1

1. newStep: the new step at which the incident has to be processed

Returns: nothing

Selected Logical View Report

Logical View

Identifier

This class is used to generate a new unique identifier for concepts that are being added to the graph. It has only one method, and one static integer variable.

Private Properties:

unique : int = 0

This is a static integer that holds the value for the next unique identifier. It keeps monotonically increasing.

Public Methods:

getNextIdentifier () : int

This method increments the unique value and returns it. This way, it maintains a monotonically increasing counter.

Arguments: 0

Returns: the next unique identifier.

Selected Logical View Report

Logical View

QueueOperations

This class contains a set of static operations on queues. These operations are used for managing the various functions on queues that are needed during simulation.

Public Methods:

merge (queue1 : Vector, queue2 : Vector) : void

This method is used to merge two queues in a time ascending order. Both the queues hold incidents, and these incidents have a time and step at which they have to be scheduled.

Arguments: 2

1. queue1: the queue that contains incidents to be merged to queue2
2. queue2: the queue that stores all the incidents in time ascending order

Returns: nothing

select (queue1 : Vector, time : int, step : int, queue2 : Vector) : void

This method selects incidents from queue1 that have to be scheduled at a particular time and step, and returns these incidents in queue2.

Arguments: 4

1. queue1: the queue from which incidents have to be selected

2. time: the time for which incidents have to be selected
3. step: the step for which incidents have to be selected
4. queue2: the queue containing the selected incidents

Returns: nothing

findPosition (queue : Vector, time : int, step : int) : int

This method finds the position at which a new incident may be inserted into a queue, so that the time ascending order is maintained. This method is supplied a queue, a time and a step, and returns the position in the queue at which the incident may be inserted.

Arguments: 3
1. queue: the queue in which a new incident has to be inserted
2. time: the time of processing of the new incident
3. step: the step of processing of the new incident

Returns: the position at which the incident may be inserted

/* NOTE: the simulator now uses the findPositionAndAdd process, as it combines the job of finding the position and inserting into the queue */

updateTimeStep (queue : Vector, time : int, step : int) : void

This method is called to update the time and step of the incidents in the suspended queue. The method takes in a queue, and the time and step to be added to the

existing time and step of the incidents. It then adds the time and step increase to all the incidents in the queue.

Arguments: 3
1. queue: the queue in which the incidents have to be updated
2. newTime: the increase in the time
3. newStep: the increase in the step

Returns: nothing

moveIncidents (queue1 : Vector, c : Concept, queue2 : Vector) : int
This method is called when an action needs to be suspended or resumed. This method takes in 2 queues and a concept, and moves all incidents from queue1 that are caused by the concept to queue2. It returns the number of such incidents moved from queue1 to queue2.

Arguments:3
1. queue1: the queue from which incidents have to be moved
2. c: the concept which causes the incidents that have to be moved
3. queue2: the queue to which the incidents have to be moved

Returns: the number of incidents moved from queue1 to queue2

terminate (queue : Vector, c : Concept) : int

This method is called when an action needs to be terminated. This method takes in a queue and a concept, and removes all incidents from the queue that are caused by the concept. It returns the number of such incidents removed.

Arguments:2

1. queue: the queue from which incidents have to be removed
2. c: the concept which causes the incidents that have to be removed

Returns: the number of incidents removed

findPositionAndAdd (queue : Vector, newIncident : Incident) : void

This method finds the position at which a new incident may be inserted into a queue, so that the time ascending order is maintained. It then inserts the incident into the queue at this position. This method is supplied a queue and the incident to be inserted.

Arguments: 2

1. queue: the queue in which a new incident has to be inserted
2. newIncident: the incident to be inserted in the queue.

Returns: nothing

Selected Logical View Report

Logical View

ActionConcept

This class has been created as a subclass of notio.Concept, specialized for action concepts. The purpose for creating this class is that it can contain the activity of the action. It also helps in generating action incidents.

Public Properties:

INITIATE : int = 1

This is a static final variable to indicate the initiation of an action.

SUSPEND : int = 2

This is a static final variable to indicate the suspension of an action.

RESUME : int = 3

This is a static final variable to indicate the resumption of an action.

TERMINATE : int = 4

This is a static final variable to indicate the termination of an action.

Private Properties:

ident : int

The identifier for the concept, this is stores which action concept this concept refers to.

type : String = "action"

This stores the type of the concept, whether it is an action, device, event, state or value. It cannot be changed.

actionType : int = 0

This attribute stores the action type, where it is initiated, suspended, resumed or terminated.

Public Methods:

getIdentifier () : int

This method is used to get the identifier of the concept. from this, we may identify the actual concept in the graph that this concept refers to.

Arguments: 0

Returns: an integer that holds the identifier

getConceptType () : String

This method returns the type of the concept, whether it is an action, device, event, state or value concept.

Arguments: 0

Returns: a string that holds the type of the concept

getActionType () : int

This method is used to access the action type of a concept. It returns an integer which tells whether the

action was initiated, suspended, resumed or terminated.

Arguments: 0

Returns: an integer holding the action type

Selected Logical View Report

Logical View

DeviceConcept

This class has been created as a subclass of notio.Concept, specialized for device concepts. The purpose for creating this class is that it helps in creating schema graphs for devices, such as registers, memory and processors.

Private Properties:

ident : int

The identifier for the concept, this is stores which device concept this concept refers to.

type : String = "device"

This stores the type of the concept, whether it is an action, device, event, state or value. It cannot be changed.

Public Methods:

getIdentifier () : int

This method is used to get the identifier of the concept. from this, we may identify the actual concept in the graph that this concept refers to.

Arguments: 0

Returns: an integer that holds the identifier

getConceptType () : String

This method returns the type of the concept, whether it is an action, device, event, state or value concept.

Arguments: 0

Returns: a string that holds the type of the concept

Selected Logical View Report

Logical View

EventConcept

This class has been created as a subclass of notio.Concept, specialized for event concepts. The purpose for creating this class is that it can help in generating event incidents.

Private Properties:

ident : int

The identifier for the concept, this is stores which event concept this concept refers to.

type : String = "event"

This stores the type of the concept, whether it is an action, device, event, state or value. It cannot be changed.

Public Methods:

getIdentifier () : int

This method is used to get the identifier of the concept. from this, we may identify the actual concept in the graph that this concept refers to.

Arguments: 0

Returns: an integer that holds the identifier

getConceptType () : String

This method returns the type of the concept, whether it is an action, device, event, state or value concept.

Arguments: 0

Returns: a string that holds the type of the concept

Selected Logical View Report

Logical View

StateConcept

This class has been created as a subclass of notio.Concept, specialized for state concepts. The purpose for creating this class is that it can hold the state of a state concept, and also help in generating state incidents.

Private Properties:

ident : int

The identifier for the concept, this is stores which state concept this concept refers to.

type : String = "state"

This stores the type of the concept, whether it is an action, device, event, state or value. It cannot be changed.

state : String = ""

This attribute stores the state of the state concept.

Public Methods:

getIdentifier () : int

This method is used to get the identifier of the concept. from this, we may identify the actual concept in the graph that this concept refers to.

Arguments: 0

Returns: an integer that holds the identifier

getConceptType () : String
This method returns the type of the concept, whether it is an action, device, event, state or value concept.

Arguments: 0

Returns: a string that holds the type of the concept

getState () : String
This method is used to access the state of the state concept.

Arguments: 0

Returns: a string that holds the state of the state concept

setState (newState : String) : void
This method is used to store a new state in the concept.

Arguments: 1
1. newState: a string that holds the new state to be stored

Returns: nothing

Selected Logical View Report

Logical View

ValueConcept

This class has been created as a subclass of notio.Concept, specialized for value concepts. The purpose for creating this class is that it can hold the value of a value concept, and also help in generating value incidents.

Private Properties:

ident : int

The identifier for the concept, this is stores which value concept this concept refers to.

type : String = "value"

This stores the type of the concept, whether it is an action, device, event, state or value. It cannot be changed.

value : int = -1

This attribute holds the value of the value concept.

Public Methods:

getIdentifier () : int

This method is used to get the identifier of the concept. from this, we may identify the actual concept in the graph that this concept refers to.

Arguments: 0

Returns: an integer that holds the identifier

getConceptType () : String
This method returns the type of the concept, whether it is an action, device, event, state or value concept.

Arguments: 0

Returns: a string that holds the type of the concept

read () : int
This method is used to read the value stored in the concept.

Arguments: 0

Returns: an integer that holds the value in the concept

write (newValue : int) : void
This method sets the value stored in the concept.

Arguments: 1
1. newValue: the new value to be stored

Returns: nothing

Selected Logical View Report

Logical View

SimulatorErrorHandler

The SimulatorErrorHandler class is used to log different errors to the log file. This class contains a set of static procedures, one for each error. These proecdures may be called from within the simulator, with parameters to specify where the error occurred.

There are three severity levels, error, warning and note.

Each of these procedures is overloaded to allow for future developers to change the severity levels of errors.

Private Properties:

mType[] : String = {"ERROR : ", "Warning: ", "Note: "}
This string array is used to display a standard message, as a result of simulation. There are three levels of messages, error, warning and note. Each error may come with a specific type of message level. In case an error arrives with no specific level, the default level is used.

rType[] : = {"SIMULATION ABORTED !", "Lack of information.", "."}
This string array is used to display a standard result, as a result of simulation. There are three levels of results, error, warning and note. Each error may come with a specific type of message level. In

case an error arrives with no specific level, the default level is used.

defaultType : int = 0
This integer is used to store the default level of the message and result to be displayed. The default level is set to zero.

Public Methods:

missingRelation (module : String, relation : String) : void
This method is called whenever a relation is missing. For a missing essential relation, the level is set to error. For a missing non-essential relation, the level is set to warning. It logs the missing relation and the module that this error was encountered in.

Arguments: 2
1. module: module in which this error was encountered
2. relation: the relation that is present

Returns: nothing

unsuitableConceptType (module : String, conceptType : String, relation : String) : void
This method is called whenever an argument of a relation does not match the concept type that the relation expects. It accepts 3 parameters, the module, concept type, and relation, and logs the fact that relation did not expect a concept of the supplied concept type as it argument in the given module.

Arguments: 3
1. module: module in which this error was encountered

2. conceptType: the concept type that is present
3. relation: the relation that is present

Returns: nothing

incompatibleSource (module : String, conceptType : string) : void
This method is called when the source of an action does not match the action being performed.

Arguments:1
1. module: module in which this error was encountered

Returns: nothing

incompatibleOperand (module : String, conceptType : String, relation : String) : void
This method is called when the operand of an action does not match the action being performed.

Arguments: 3
1. module: module in which this error was encountered
2. conceptType: the erroneous concept type
3. relation: the relation which is the cause for error

Returns: nothing

unknownOperation (module : String, operation : String) : void
This method is called when the action has to be perform an unknown operation.

Arguments: 2

1. module: the module in which this error was encountered
2. operation: the operation that is unknown

Returns: nothing

unknownActionType (action : String) : void
This method is called when an unknown action type is encountered. There are only 4 action types, initiate, suspend, resume, and terminate.

Arguments: 1
1. action: the action for which this unknown type occurred

Returns: nothing.

unknownAction (action : String) : void
This method is called when an action that does not have an associated procedure is encountered.

Arguments: 1
1. action: the action that does not have an action procedure

Returns: nothing.

illegalActionType (action : String, act : String) : void
This method is called when an illegal action type is encountered. For example, if an action is not running, and it is terminated, or suspended, that is an error.

Arguments: 2

1. action: the action module where this happened
2. act: the current activity of the action

Returns: nothing

unableToSimulateInnerGraph (concept : String) : void
This method id invoked when an inner graph is not properly simulated. The state concept for which this occurred is taken as a parameter.

Arguments: 1
1. concept: the concept for which the inner graph could not be simulated.

Returns: nothing

unknownDevice (module : String, device : String) : void
This method is invoked when an unknown device is encountered, and cannot be expanded.

Arguments: 2
1. module: the module in which this occurring to it.
2. concept: the concept that could not be elaborated

Returns: nothing

Appendix E – Test Cases and Results

This appendix lists the test cases used to test the simulator, as well as the simulation results. They may be used to understand the way in which the simulator deals with the different types of concepts and relations.

Test Case 1 <negatereg.cgf>

Objective: To test that the simulator accepts an input file and is able to extract the graph from the file and simulate it

```
[Proposition:''
[action:*a'negate']
[device:*b'register_reg1']
[value:*c'v_5']
(operand?a?b)
(result?a?b)
(delay?a?c)]
```

Expected Result: The simulator logs the date and time of the simulation, the file name being simulated, and the number of concepts and relations in the test file.

Simulator Output:

```
Fri Apr 20 17:25:47 EDT 2001
File simulated: negatereg.cgf

Graph has 3 concepts and 3 relations.

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 2 <negateval.cgf>

Objective: To test that the simulator is capable of performing a simple negation action using only value concepts

```
[Proposition:''
[action:*a'negate']
[value:*b'v_25']
[value:*c'v_5']
(operand?a?b)
(result?a?b)
(delay?a?c)]
```

Expected Result: The simulator logs the date and time of the simulation, the file name being simulated, and the number of concepts and relations in the test file. It then negates the value of the value concept and stores it back within the value concept. Also, the activity of the action remains active for the amount of time mentioned in the file.

Simulator Output:

```
Thu Apr 26 19:31:44 EDT 2001
File simulated: negateval.cgf

Graph has 3 concepts and 3 relations.

TIME: [0]
     STEP: [0]
     STEP: [1]
          ACTION: negate (mode = active)

TIME: [5]
     STEP: [0]
          VALUE: v_25 (eval = '-25')
```

ACTION: negate (mode = inactive)

FILE SIMULATION DONE!

Conclusion: The simulator output matches the expected result.

****

Test Case 3 <negateval.cgf>

Objective: To test that the simulator is capable of performing a simple negation action using only value concepts, with the “more” output option. This output option should show each concept in the graph being prepared for simulation.

[Proposition:''
[action:*a'negate']
[value:*b'v_25']
[value:*c'v_5']
(operand?a?b)
(result?a?b)
(delay?a?c)]

Expected Result: The simulator logs the date and time of the simulation, the file name being simulated, and the number of concepts and relations in the test file. It then prepares each concept in the graph and lists the details of the preparation of the graph. Details of all the queue and incidents are also listed in the output. The *negate* action negates the value of the value concept and stores it back within the value concept. Also, the activity of the action remains active for the amount of time mentioned in the file.

Simulator Output:

Thu Apr 26 19:32:31 EDT 2001

```
File simulated: negateval.cgf

Graph has 3 concepts and 3 relations.
      in prepare
            First preparing all value state and action concepts
            concept has no coref sets
            IT DID COME IN HERE
            IT DID COME OUT HERE TOO
            now handling a action concept.
            concept negate is an action concept
            adding a state concept with ident 4
            Graph has 4 C, 4 R.

            concept has no coref sets
            IT DID COME IN HERE
            IT DID COME OUT HERE TOO
            now handling a value concept.
            concept v_25 is a value concept
            This is a value concept present before device
preparation
            Therefore a new ValueConcept is added  with the eval
relation
            Graph has 5 C, 5 R.

            concept has no coref sets
            IT DID COME IN HERE
            IT DID COME OUT HERE TOO
            now handling a value concept.
            concept v_5 is a value concept
            This is a value concept present before device
preparation
            Therefore a new ValueConcept is added  with the eval
relation
            Graph has 6 C, 6 R.
```

```
      The graph has been prepared
            Now preparing all device concepts
            concept has no coref sets
            Graph has 6 C, 6 R.

            concept has no coref sets
            Graph has 6 C, 6 R.

            concept has no coref sets
            Graph has 6 C, 6 R.

      Now ready for simulation
Initial Incident vector of size 1 received.
Graph: 6 C, 6 R.

TIME: [0]
      STEP: [0]
      After select, IQ and CQ sizes are 0 and 1
      Found 0 value incidents.
      Found 0 state incidents.
      Found 0 event incidents.
      Found 1 action incidents.
            For action incident 1
            Concept is: negate
            concept has no coref sets
            action is being initiated
      Before merge, NQ and IQ sizes are 1 and 0
      After merge, IQ size is 1
      STEP: [1]
      After select, IQ and CQ sizes are 0 and 1
      Found 0 value incidents.
      Found 1 state incidents.
            For state incident 1
```

```
            ACTION: negate (mode = active)
            Firing for concept negate
      Found 0 event incidents.
      Found 0 action incidents.
      Before merge, NQ and IQ sizes are 2 and 0
      After merge, IQ size is 2

TIME: [5]
      STEP: [0]
      After select, IQ and CQ sizes are 0 and 2
      Found 1 value incidents.
            For value incident 1
            VALUE: v_25 (eval = '-25')
      Found 1 state incidents.
            For state incident 1
            ACTION: negate (mode = inactive)
            Firing for concept negate
      Found 0 event incidents.
      Found 0 action incidents.
      Before merge, NQ and IQ sizes are 0 and 0
      After merge, IQ size is 0

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 4 <negatereg.cgf>

Objective: To test that the simulator is capable of performing a simple negation action using device concepts. The “more” output option is used. This output option should show each concept in the graph, including device concepts, being prepared for simulation.

```
[Proposition:''
[action:*a'negate']
[device:*b'register_reg1']
[value:*c'v_5']
(operand?a?b)
(result?a?b)
(delay?a?c)]
```

Expected Result: The simulator logs the date and time of the simulation, the file name being simulated, and the number of concepts and relations in the test file. It then prepares each concept in the graph (including the device concepts) and lists the details of the preparation of the graph. Details of all the queue and incidents are also listed in the output. The *negate* action negates the value of the value of the register and stores it back within the device concept. Also, the activity of the action remains active for the amount of time mentioned in the file.

Simulator Output:

```
Thu Apr 26 19:33:12 EDT 2001
File simulated: negatereg.cgf

Graph has 3 concepts and 3 relations.
      in prepare
            First preparing all value state and action concepts
            concept has no coref sets
            IT DID COME IN HERE
            IT DID COME OUT HERE TOO
            now handling a action concept.
            concept negate is an action concept
            adding a state concept with ident 4
            Graph has 4 C, 4 R.
```

```
            concept has no coref sets
            IT DID COME IN HERE
            IT DID COME OUT HERE TOO
            now handling a device concept.
            THIS MEANS IT IS A DEVICE
            Graph has 4 C, 4 R.

            concept has no coref sets
            IT DID COME IN HERE
            IT DID COME OUT HERE TOO
            now handling a value concept.
            concept v_5 is a value concept
            This is a value concept present before device
preparation
            Therefore a new ValueConcept is added with the eval
relation
            Graph has 5 C, 5 R.

      The graph has been prepared
            Now preparing all device concepts
            concept has no coref sets
            Graph has 5 C, 5 R.

            concept has no coref sets
            concept register_reg1 is a device concept
            The device is a register
            Register joined with schema graph successfully
            Graph has 8 C, 8 R.

            concept has no coref sets
            Graph has 8 C, 8 R.

      Now ready for simulation
```

```
Initial Incident vector of size 1 received.
Graph: 8 C, 8 R.

TIME: [0]
      STEP: [0]
      After select, IQ and CQ sizes are 0 and 1
      Found 0 value incidents.
      Found 0 state incidents.
      Found 0 event incidents.
      Found 1 action incidents.
            For action incident 1
            Concept is: negate
            concept has no coref sets
            action is being initiated
      Before merge, NQ and IQ sizes are 1 and 0
      After merge, IQ size is 1
      STEP: [1]
      After select, IQ and CQ sizes are 0 and 1
      Found 0 value incidents.
      Found 1 state incidents.
            For state incident 1
            ACTION: negate (mode = active)
            Firing for concept negate
      Found 0 event incidents.
      Found 0 action incidents.
      Before merge, NQ and IQ sizes are 2 and 0
      After merge, IQ size is 2

TIME: [5]
      STEP: [0]
      After select, IQ and CQ sizes are 0 and 2
      Found 1 value incidents.
            For value incident 1
            DEVICE: register_reg1 (contain = '-1')
```

```
      Found 1 state incidents.
            For state incident 1
            ACTION: negate (mode = inactive)
            Firing for concept negate
      Found 0 event incidents.
      Found 0 action incidents.
      Before merge, NQ and IQ sizes are 0 and 0
      After merge, IQ size is 0

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 5 <transferval.cgf>

Objective: To test that the simulator is capable of processing propagational relations such as *initiate*, *suspend* and *resume*.

```
[Proposition:''
[action:*a'transfer']
[value:*b'v_12']
[value:*c'v_25']
[value:*d'v_15']
[value:*e'v_1']
(source?a?b)
(destination?a?c)
(delay?a?d)
(quant?a?e)
[action:*f'sleep']
[value:*g'v_5']
[action:*h'wake']
(initiate?a?f)
(delay?f?g)
```

(suspend_t?f?a)
(initiate_t?f?h)
(delay?h?g)
(resume_t?h?a)]

Expected Result: The transfer action is initiated by the set of initial incidents. This transfer action should initiate the sleep action, which suspends the transfer when it completes execution. The sleep initiates the wake action. The wake action resumes the transfer when it completes execution.

Simulator Output:

```
Thu Apr 26 19:33:52 EDT 2001
File simulated: transferval.cgf

Graph has 8 concepts and 10 relations.

TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: transfer (mode = active)
      STEP: [2]
      STEP: [3]
            ACTION: sleep (mode = active)

TIME: [5]
      STEP: [0]
            ACTION: sleep (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: wake (mode = active)
            ACTION: transfer (mode = suspended)
```

```
TIME: [10]
      STEP: [0]
            ACTION: wake (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: transfer (mode = active)

TIME: [20]
      STEP: [0]
      STEP: [2]
            VALUE: v_25 (eval = '12')
            ACTION: transfer (mode = inactive)

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 6 <loadandtransferreg.cgf>

Objective: To test that the simulator is capable of processing a series of steps indicated the repeated occurrences of the *initiate_t* relation. The test case loads two registers with initial values of 17 and 23 and then performs a transfer action.

```
[Proposition:''
[action:*a'write']
[device:*b'register_reg1']
[value:*c'v_17']
[value:*d'v_10']
[action:*e'write']
[device:*f'register_reg2']
[value:*g'v_23']
[action:*h'transfer']
```

```
[value:*i'v_1']
(source?a?c)
(destination?a?b)
(delay?a?d)
(initiate_t?a?e)
(source?e?g)
(destination?e?f)
(delay?e?d)
(initiate_t?e?h)
(source?h?b)
(destination?h?f)
(delay?h?d)
(quant?h?i)]
```

Expected Result: The write action loads one register with a value of 17, and causes the other write, which loads the second register with a value of 23. The transfer action is initiated by the second write. This transfer action should transfer a value of 17 to the second register.

Simulator Output:

```
Thu Apr 26 19:34:50 EDT 2001
File simulated: loadandtransferreg.cgf

Graph has 9 concepts and 12 relations.

TIME: [0]
     STEP: [0]
     STEP: [1]
          ACTION: write (mode = active)

TIME: [10]
     STEP: [0]
          DEVICE: register_reg1 (contain = '17')
```

```
            ACTION: write (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: write (mode = active)

TIME: [20]
      STEP: [0]
            DEVICE: register_reg2 (contain = '23')
            ACTION: write (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: transfer (mode = active)

TIME: [30]
      STEP: [0]
            DEVICE: register_reg2 (contain = '17')
            ACTION: transfer (mode = inactive)

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 7 <loadandmultiplyreg.cgf>

Objective: To test that the simulator is capable of processing a series of steps indicated the repeated occurrences of the *initiate_t* relation. The test case loads two registers with initial values of 17 and 23. This test also checks on the ability of the simulator to perform mathematical operations like multiplication.

```
[Proposition:''
[action:*a'write']
```

[device:*b'register_reg1']
[device:*c'register_reg2']
[value:*d'v_15']
[value:*e'v_17']
[value:*f'v_23']
[device:*g'register_reg3']
[action:*h'write']
[action:*i'multiply']
(source?a?e)
(destination?a?b)
(delay?a?d)
(initiate_t?a?h)
(source?h?f)
(destination?h?c)
(delay?h?d)
(initiate_t?h?i)
(operand1?i?b)
(operand2?i?c)
(result?i?g)
(delay?i?d)]

Expected Result: The write action loads one register with a value of 17, and causes the other write, which loads the second register with a value of 23. The multiply action is initiated by the second write. This multiply action should multiply the 2 values and store the product in the third register.

Simulator Output:
Thu Apr 26 19:35:25 EDT 2001
File simulated: loadandmultiplyreg.cgf

Graph has 9 concepts and 12 relations.

TIME: [0]

```
      STEP: [0]
      STEP: [1]
            ACTION: write (mode = active)

TIME: [15]
      STEP: [0]
            DEVICE: register_reg1 (contain = '17')
            ACTION: write (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: write (mode = active)

TIME: [30]
      STEP: [0]
            DEVICE: register_reg2 (contain = '23')
            ACTION: write (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: multiply (mode = active)

TIME: [45]
      STEP: [0]
            DEVICE: register_reg3 (contain = '391')
            ACTION: multiply (mode = inactive)

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 8 <eventfire.cgf>

Objective: To test that the simulator uses the fired event queue correctly. This test case fires two events and then initiates an action only if both these events have been previously fired.

```
[Proposition:''
[action:*a'wake']
[event:*b'e1']
[event:*c'e2']
[action:*d'sleep']
[value:*e'v_20']
[state:*f'condition'
     [event:?b]
     [event:?c]
     (and?b?c)]
(initiate?a?d)
(w_if?a?f)
(delay?a?e)
(delay?d?e)]
```

Expected Result: The wake action triggers both events when it completes execution. It also initiates the sleep action if both the events have been fired. The expected output is that the sleep action gets initiated.

Simulator Output:

```
Thu Apr 26 19:36:29 EDT 2001
File simulated: eventfire.cgf

Graph has 6 concepts and 4 relations.

TIME: [0]
     STEP: [0]
     STEP: [1]
          ACTION: wake (mode = active)
```

```
      STEP: [2]
      STEP: [3]
            ACTION: sleep (mode = active)

TIME: [20]
      STEP: [0]
            ACTION: wake (mode = inactive)
            ACTION: sleep (mode = inactive)

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 9 <loadandexponentiatemem.cgf>

Objective: To test that the simulator is able to expand a memory device concept, and perform the operation of exponentiation on two operands fetched from memory, and to write the result back to memory.

```
[Proposition:''
[action:*a'write']
[device:*b'memory_RAM']
[value:*c'v_3']
[value:*d'v_15']
[value:*e'v_4']
[value:*f'v_5']
[value:*g'v_1']
[action:*h'write']
[action:*i'exponentiate']
[value:*j'v_6']
(source?a?e)
(destination?a?b)
```

(target?a?c)
(delay?a?d)
(initiate_t?a?h)
(source?h?f)
(destination?h?b)
(delay?h?d)
(target?h?j)
(initiate_t?h?i)
(operand1?i?b)
(location1?i?c)
(operand2?i?b)
(location2?i?j)
(result?i?b)
(target?i?g)
(delay?i?d)]

Expected Result: The simulator should correctly expand the device concept for memory, and initialize all the memory locations to 0. Then, two memory locations are loaded with values from value concepts, which are then used as operands by the exponentiate action, and the result should get written back to another location in memory.

Simulator Output:

```
Thu Apr 26 19:37:42 EDT 2001
File simulated: loadandexponentiatemem.cgf

Graph has 10 concepts and 17 relations.

TIME: [0]
     STEP: [0]
     STEP: [1]
          ACTION: write (mode = active)
```

```
TIME: [15]
      STEP: [0]
            DEVICE: memory_RAM (containAt = '2')
            ACTION: write (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: write (mode = active)

TIME: [30]
      STEP: [0]
            DEVICE: memory_RAM (containAt = '16')
            ACTION: write (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: exponentiate (mode = active)

TIME: [45]
      STEP: [0]
            DEVICE: memory_RAM (containAt = '65536')
            ACTION: exponentiate (mode = inactive)

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 10 <states.cgf>

Objective: To test that the simulator is able to execute nested graphs and evaluate states that may be a combination of nested graphs. The test is setup so that action1 initiates action2 if the enabling condition is true. The enabling condition is the logical OR of two facts; the first is that action1 is active, and the second is that 2344 >= 12433. This test is run with “medium”

output option to ensure that all the internal processing details are visible.

```
[Proposition:''
[action:*a'sleep']
[action:*b'wake']
[value:*c'v_20']
[state:*d's_false'
     [state:*e'condition1'
          [action:?a]
          [state:*f's_active']
          (eq?a?f)]
     [state:*g'condition2'
          [value:*h'v_2344']
          [value:*i'v_12433']
          (ge?h?i)]
     (w_or?e?g)]
(w_if?a?d)
(initiate?a?b)
(delay?a?c)
(delay?b?c)]
```

Expected Result: The simulator should begin by initiating action1. Action1 should evaluate the enabling condition, find that it is true, and initiate action2. Here action1 is represented by sleep, and action2 is represented by wake. The result is that the wake action should be initiated.

Simulator Output:

```
Thu Apr 26 19:38:31 EDT 2001
File simulated: states.cgf

Graph has 4 concepts and 4 relations.
          adding a state concept with ident 5
```

```
            adding a state concept with ident 7
            concept 3 is a context
            concept 0 is a context
            concept 1 is a context
Initial Incident vector of size 1 received.

TIME: [0]
      STEP: [0]
      Found 0 value incidents.
      Found 0 state incidents.
      Found 0 event incidents.
      Found 1 action incidents.
            For action incident 1
            action is being initiated
      STEP: [1]
      Found 0 value incidents.
      Found 1 state incidents.
            For state incident 1
            ACTION: sleep (mode = active)
                  processing an AND or OR relation in evaluate
                  processing an EQ or NE relation in evaluate
                  The first concept is an action concept.
                  Its mode is: active
                  The second concept is a state concept.
                  Its state is: active
                  The final result is: true
                  First value is: 'true'
                  processing an arithmetic compare in evaluate
                  First value is: 2344
                  Second value is: 12433
                  The final result is: false
                  Second value is: 'false'
                  The final result is: true
            sleep initiates action WAKE
```

```
      Found 0 event incidents.
      Found 0 action incidents.
      STEP: [2]
      Found 0 value incidents.
      Found 0 state incidents.
      Found 0 event incidents.
      Found 1 action incidents.
            For action incident 1
            action is being initiated
      STEP: [3]
      Found 0 value incidents.
      Found 1 state incidents.
            For state incident 1
            ACTION: wake (mode = active)
      Found 0 event incidents.
      Found 0 action incidents.

TIME: [20]
      STEP: [0]
      Found 0 value incidents.
      Found 2 state incidents.
            For state incident 1
            ACTION: sleep (mode = inactive)
                  processing an AND or OR relation in evaluate
                  processing an EQ or NE relation in evaluate
                  The first concept is an action concept.
                  Its mode is: inactive
                  The second concept is a state concept.
                  Its state is: active
                  The final result is: false
                  First value is: 'false'
                  processing an arithmetic compare in evaluate
                  First value is: 2344
                  Second value is: 12433
```

```
                The final result is: false
                Second value is: 'false'
                The final result is: false
          enable condition exists, action not enabled, returning
          For state incident 2
          ACTION: wake (mode = inactive)
     Found 0 event incidents.
     Found 0 action incidents.

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 11 <valueexample.cgf>

Objective: To test a graph that models an iterative action using the *initiate_t* relation and an enabling condition. The test setup initializes a value concept to hold the value 1. This value is incremented until it reaches a value of 7. At this point counting is stopped.

```
[Proposition:''
[action:*a'increment']
[value:*b'v_10']
[value:*c'v_1']
[value:*d'data']
[state:*e''
     [value:*f'v_7']
     [value:?d'']
     (lt?d?f)]
(initiate_t?a?a)
(delay?a?b)
(operand1?a?c)
```

(operand?a?d)
(result?a?d)
(w_if?a?e)]

Expected Result: The increment action initiates itself for six times, and the value is incremented until it reaches a value of 7. Incrementing is stopped at this point.

Simulator Output:

```
Thu Apr 26 19:39:06 EDT 2001
File simulated: valueexample.cgf

Graph has 6 concepts and 9 relations.

TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: write (mode = active)

TIME: [10]
      STEP: [0]
            VALUE: data (eval = '1')
            ACTION: write (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)

TIME: [20]
      STEP: [0]
            VALUE: data (eval = '2')
            ACTION: increment (mode = inactive)
      STEP: [1]
      STEP: [2]
```

```
            ACTION: increment (mode = active)

TIME: [30]
      STEP: [0]
            VALUE: data (eval = '3')
            ACTION: increment (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)

TIME: [40]
      STEP: [0]
            VALUE: data (eval = '4')
            ACTION: increment (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)

TIME: [50]
      STEP: [0]
            VALUE: data (eval = '5')
            ACTION: increment (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)

TIME: [60]
      STEP: [0]
            VALUE: data (eval = '6')
            ACTION: increment (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)
```

```
TIME: [70]
      STEP: [0]
            VALUE: data (eval = '7')
            ACTION: increment (mode = inactive)

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 12 <newfact.cgf>

Objective: To test a graph that models an iterative algorithm for factorial generation. The 3 register concepts are first loaded with ones. The repeated calling of the increment, multiply and transfer procedures is performed till the value whose factorial is needed is found.

```
[Proposition:''
[action:*a'write']
[action:*b'write']
[action:*c'write']
[value:*d'v_10']
[value:*e'v_25']
[device:*f'register_number']
[device:*g'register_oldprod']
[device:*h'register_newprod']
[value:*i'v_1']
[state:*j'condition'
      [device:?f'']
      [value:*k'v_9']
      (lt?f?k)]
[value:*l'v_15']
```

```
[action:*m'transfer']
[action:*n'increment']
[action:*o'multiply']
[value:*p'v_1']

(source?a?p)
(destination?a?f)
(delay?a?d)
(initiate_t?a?b)

(source?b?p)
(destination?b?g)
(delay?b?d)
(initiate_t?b?c)

(source?c?p)
(destination?c?h)
(delay?c?d)
(initiate_t?c?m)

(source?m?h)
(destination?m?g)
(delay?m?l)
(initiate_t?m?n)
(enable_if?m?j)

(operand?n?f)
(result?n?f)
(delay?n?d)
(initiate_t?n?o)

(operand1?o?f)
(operand2?o?g)
(result?o?h)
```

(delay?o?e)
(initiate_t?o?m)]

Expected Result: The simulator continues incrementing, multiplying and transferring values till the factorial is calculated.

Simulation Output:

```
Thu Apr 26 19:40:43 EDT 2001
File simulated: newfact.cgf

Graph has 15 concepts and 26 relations.

TIME: [0]
     STEP: [0]
     STEP: [1]
          ACTION: write (mode = active)

TIME: [10]
     STEP: [0]
          DEVICE: register_number (contain = '1')
          ACTION: write (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: write (mode = active)

TIME: [20]
     STEP: [0]
          DEVICE: register_oldprod (contain = '1')
          ACTION: write (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: write (mode = active)
```

```
TIME: [30]
     STEP: [0]
          DEVICE: register_newprod (contain = '1')
          ACTION: write (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: transfer (mode = active)

TIME: [45]
     STEP: [0]
          DEVICE: register_oldprod (contain = '1')
          ACTION: transfer (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: increment (mode = active)

TIME: [55]
     STEP: [0]
          DEVICE: register_number (contain = '2')
          ACTION: increment (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: multiply (mode = active)

TIME: [80]
     STEP: [0]
          DEVICE: register_newprod (contain = '2')
          ACTION: multiply (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: transfer (mode = active)

TIME: [95]
     STEP: [0]
```

```
            DEVICE: register_oldprod (contain = '2')
            ACTION: transfer (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)

TIME: [105]
      STEP: [0]
            DEVICE: register_number (contain = '3')
            ACTION: increment (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: multiply (mode = active)

TIME: [130]
      STEP: [0]
            DEVICE: register_newprod (contain = '6')
            ACTION: multiply (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: transfer (mode = active)

TIME: [145]
      STEP: [0]
            DEVICE: register_oldprod (contain = '6')
            ACTION: transfer (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)

TIME: [155]
      STEP: [0]
            DEVICE: register_number (contain = '4')
            ACTION: increment (mode = inactive)
```

```
      STEP: [1]
      STEP: [2]
            ACTION: multiply (mode = active)

TIME: [180]
      STEP: [0]
            DEVICE: register_newprod (contain = '24')
            ACTION: multiply (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: transfer (mode = active)

TIME: [195]
      STEP: [0]
            DEVICE: register_oldprod (contain = '24')
            ACTION: transfer (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)

TIME: [205]
      STEP: [0]
            DEVICE: register_number (contain = '5')
            ACTION: increment (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: multiply (mode = active)

TIME: [230]
      STEP: [0]
            DEVICE: register_newprod (contain = '120')
            ACTION: multiply (mode = inactive)
      STEP: [1]
      STEP: [2]
```

```
            ACTION: transfer (mode = active)

TIME: [245]
      STEP: [0]
            DEVICE: register_oldprod (contain = '120')
            ACTION: transfer (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)

TIME: [255]
      STEP: [0]
            DEVICE: register_number (contain = '6')
            ACTION: increment (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: multiply (mode = active)

TIME: [280]
      STEP: [0]
            DEVICE: register_newprod (contain = '720')
            ACTION: multiply (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: transfer (mode = active)

TIME: [295]
      STEP: [0]
            DEVICE: register_oldprod (contain = '720')
            ACTION: transfer (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)
```

```
TIME: [305]
     STEP: [0]
          DEVICE: register_number (contain = '7')
          ACTION: increment (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: multiply (mode = active)

TIME: [330]
     STEP: [0]
          DEVICE: register_newprod (contain = '5040')
          ACTION: multiply (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: transfer (mode = active)

TIME: [345]
     STEP: [0]
          DEVICE: register_oldprod (contain = '5040')
          ACTION: transfer (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: increment (mode = active)

TIME: [355]
     STEP: [0]
          DEVICE: register_number (contain = '8')
          ACTION: increment (mode = inactive)
     STEP: [1]
     STEP: [2]
          ACTION: multiply (mode = active)

TIME: [380]
     STEP: [0]
```

```
            DEVICE: register_newprod (contain = '40320')
            ACTION: multiply (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: transfer (mode = active)

TIME: [395]
      STEP: [0]
            DEVICE: register_oldprod (contain = '40320')
            ACTION: transfer (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: increment (mode = active)

TIME: [405]
      STEP: [0]
            DEVICE: register_number (contain = '9')
            ACTION: increment (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: multiply (mode = active)

TIME: [430]
      STEP: [0]
            DEVICE: register_newprod (contain = '362880')
            ACTION: multiply (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: transfer (mode = active)
      STEP: [3]
            ACTION: transfer (mode = inactive)

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 13 <missingrelation.cgf>

Objective: This test case shows the output of the simulator for the *negate* action, when the operand relation (which is an essential relation) is missing.

```
[Proposition:''
[action:*a'negate']
[value:*b'v_25']
[value:*c'v_5']
(result?a?b)
(delay?a?c)]
```

Expected Output: The simulator should end simulation, saying that the *operand* relation is missing for the *negate* action.

Simulator Output:

```
Thu Apr 26 19:43:15 EDT 2001
File simulated: missingrelation.cgf

Graph has 3 concepts and 2 relations.

TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: negate (mode = active)
ERROR   :   Procedure   'negate',   missing   relation   'operand'.
SIMULATION ABORTED!

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 14 <addstate.cgf>

Objective: This test case shows the output of the simulator for the *add* action, when one of the operands for the action is not a value concept.

```
[Proposition:''
[action:*a'add']
[value:*b'v_23']
[state:*c's_ready']
[value:*d'v_35']
[value:*e'v_10']
(operand1?a?b)
(operand2?a?c)
(result?a?d)
(delay?a?e)]
```

Expected Output: The simulator should end simulation, saying that a state concept is unsuitable as an argument for an *operand* relation, in the *add* action.

Simulator Output:

```
Thu Apr 26 19:43:54 EDT 2001
File simulated: addstate.cgf

Graph has 5 concepts and 4 relations.

TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: add (mode = active)
```

ERROR : Procedure 'add' has unsuitable concept type(s) 'state' as the argument of the relation 'operand2'. SIMULATION ABORTED!

FILE SIMULATION DONE!

Conclusion: The simulator output matches the expected result.

****

Test Case 15 <transfermultval.cgf>
Objective: This test case shows the output of the simulator for the *transfer* action, when a multiple value transfer is attempted from a single value concept.

[Proposition:''
[action:*a'transfer']
[value:*b'v_12']
[value:*c'v_25']
[value:*d'v_15']
[value:*e'v_5']
(source?a?b)
(destination?a?c)
(delay?a?d)
(quant?a?e)]

Expected Output: The simulator should end simulation, saying that the concept type is incompatible for performing the action.

Simulator Output:
Thu Apr 26 19:44:24 EDT 2001
File simulated: transfermultval.cgf

Graph has 5 concepts and 4 relations.

```
TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: transfer (mode = active)
ERROR : Procedure 'transfer', source concept incompatible for performing the action. SIMULATION ABORTED!

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 16 <illegalsuspend.cgf>

Objective: This test case shows the output of the simulator for the *transfer* action, when the action tries to *suspend* the non-active *sleep* action.

```
[Proposition:''
[action:*a'transfer']
[value:*b'v_12']
[value:*c'v_25']
[value:*d'v_15']
[value:*e'v_1']
(source?a?b)
(destination?a?c)
(delay?a?d)
(quant?a?e)
[action:*f'sleep']
(delay?f?d)
(suspend_t?a?f)]
```

Expected Output: As the *sleep* action is not running, it cannot be suspended. Therefore, the simulator should give an error to that effect and abort the simulation.

Simulator Output:

```
Thu Apr 26 19:44:50 EDT 2001
File simulated: illegalsuspend.cgf

Graph has 6 concepts and 6 relations.

TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: transfer (mode = active)

TIME: [15]
      STEP: [0]
            VALUE: v_25 (eval = '12')
            ACTION: transfer (mode = inactive)
      STEP: [1]
ERROR : Illegal action type: cannot SUSPEND action 'sleep' as it
is not ACTIVE. SIMULATION ABORTED!

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 17 <unknownaction.cgf>

Objective: This test case shows the output of the simulator for the *transfer* action, when the *transfer* action tries to initiate an action *gojump* for which no action procedure exists.

```
[Proposition:''
[action:*a'transfer']
[value:*b'v_12']
[value:*c'v_25']
[value:*d'v_15']
[value:*e'v_1']
(source?a?b)
(destination?a?c)
(delay?a?d)
(quant?a?e)
[action:*f'gojump']
[value:*g'v_5']
(initiate_t?a?f)
(delay?f?g)]
```

Expected Output: After performing the transfer, the transfer action tries to initiate action *gojump*, but since this action does not have an associated procedure, the simulator should give an error to this effect and aborts simulation.

Simulator Output:

```
Thu Apr 26 19:45:20 EDT 2001
File simulated: unknownaction.cgf

Graph has 7 concepts and 6 relations.

TIME: [0]
     STEP: [0]
     STEP: [1]
          ACTION: transfer (mode = active)

TIME: [15]
     STEP: [0]
          VALUE: v_25 (eval = '12')
```

```
            ACTION: transfer (mode = inactive)
      STEP: [1]
      STEP: [2]
            ACTION: gojump (mode = active)
ERROR : Action 'gojump' does not have a procedure associated with
it. SIMULATION ABORTED!

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 18 <unknownoperation.cgf>

Objective: This test case shows the output of the simulator for the *execute* action, when the operation to be performed is not among the operations that are supported by the command.

```
[Proposition:''
[action:*a'execute']
[value:*b'v_23']
[value:*c'v_45']
[value:*d'v_15']
[value:*e'highlight']
[value:*f'v_35']
(operand1?a?b)
(operand2?a?c)
(operation?a?e)
(result?a?f)
(delay?a?d)]
```

Expected Output: The *execute* command performs an operation on the given operands and then stores the result in another concept. When the operation to be performed is not known to the simulator,

the simulator should give an error saying that an unknown operation was encountered during simulation.

Simulator Output:

```
Thu Apr 26 19:45:56 EDT 2001
File simulated: unknownoperation.cgf

Graph has 6 concepts and 5 relations.

TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: execute (mode = active)
ERROR : Procedure 'execute', has unknown operation 'highlight'.
SIMULATION ABORTED!

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 19 <unknowndevice.cgf>

Objective: This test case shows the output of the simulator for the *multiply* action, when the operation is to be performed using device concepts. One of the device concepts is of a type that does not have a schema associated with it.

```
[Proposition:''
[action:*a'write']
[device:*b'register_reg1']
[device:*c'register_reg2']
[value:*d'v_15']
```

[value:*e'v_17']
[value:*f'v_23']
[device:*g'latch_reg3']
[action:*h'write']
[action:*i'multiply']
(source?a?e)
(destination?a?b)
(delay?a?d)
(initiate_t?a?h)
(source?h?f)
(destination?h?c)
(delay?h?d)
(initiate_t?h?i)
(operand1?i?b)
(operand2?i?c)
(result?i?g)
(delay?i?d)]

Expected Output: The simulator should first try to prepare all the device concepts in the graph. When it tries to prepare the device with a non-existing schema graph, the simulator should give an error saying that an unknown device was encountered during preparation of the graph. Also, since the initial incident has been specified to be an action incident, nothing should take place when the incident gets processed.

Simulator Output:

Thu Apr 26 19:46:31 EDT 2001
File simulated: unknowndevice.cgf

Graph has 9 concepts and 12 relations.
ERROR : in module prepare: Schema for 'latch_reg3' is not available. SIMULATION ABORTED!

```
TIME: [0]
      STEP: [0]
      Error!! Incidents of unknown type found in queue
      The type of incident 0 is: action

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 20 <innergraph.cgf>

Objective: This test case shows the output of the simulator for the *sleep* action, when the operation evaluates the inner graph for the enabling condition, and is unable to simulate and evaluate the inner graph. This test case is run with the “medium” output option, so that internal working of the simulator may also be logged to the file.

```
[Proposition:''
[action:*a'sleep']
[action:*b'wake']
[value:*c'v_20']
[state:*d'main condition'
      [state:*e'condition1'
            [action:?a]
            [state:*f's_active']
            (eq?a?f)]
      [state:*g'condition2'
            [value:*h'v_2344']
            [value:*i'v_12433']
            (ge?h?i)
            (gt?h?i)]
```

```
     (w_or?e?g)]
(w_if?a?d)
(initiate?a?b)
(delay?a?c)
(delay?b?c)]
```

Expected Output: The simulator should first initiate the *sleep* action, which tries to evaluate the enabling condition for its execution. Because of the two relations between the existing concepts, the simulator should be unable to evaluate the inner graph. Hence, the simulator should quit simulation, saying that the inner graph could not be simulated.

Simulation Output:

```
Thu Apr 26 19:47:21 EDT 2001
File simulated: innergraph.cgf

Graph has 4 concepts and 4 relations.
           adding a state concept with ident 5
           adding a state concept with ident 7
           concept 3 is a context
           concept 0 is a context
           concept 1 is a context
Initial Incident vector of size 1 received.

TIME: [0]
     STEP: [0]
     Found 0 value incidents.
     Found 0 state incidents.
     Found 0 event incidents.
     Found 1 action incidents.
          For action incident 1
          action is being initiated
     STEP: [1]
```

```
      Found 0 value incidents.
      Found 1 state incidents.
            For state incident 1
            ACTION: sleep (mode = active)
                  processing an AND or OR relation in evaluate
                  processing an EQ or NE relation in evaluate
                  The first concept is an action concept.
                  Its mode is: active
                  The second concept is a state concept.
                  Its state is: active
                  The final result is: true
                  First value is: 'true'
ERROR : Unable to simulate inner graph of concept 'condition2',
as it has more than 1 relation. SIMULATION ABORTED!
ERROR : Procedure 'sleep' has unsuitable concept type(s) 'state'
as the argument of the relation 'enable_if'. SIMULATION ABORTED!

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 21 <badoperand.cgf>

Objective: This test case shows the output of the simulator for the *sleep* action, when the operation evaluates the inner graph for the enabling condition, and the inner graph involves an operation with unsuitable operands. This test case is run with the “medium” output option, so that internal working of the simulator may also be logged to the file.

```
[Proposition:''
[action:*a'sleep']
[action:*b'wake']
```

```
[value:*c'v_20']
[state:*d'main condition'
     [state:*e'condition1'
          [action:?a]
          [state:*f's_active']
          (eq?a?f)]
     [state:*g'condition2'
          [value:*h'v_2344']
          [value:*i'v_12433']
          (and?h?i)]
     (w_or?e?g)]
(w_if?a?d)
(initiate?a?b)
(delay?a?c)
(delay?b?c)]
```

Expected Output: The simulator should first initiate the *sleep* action, which tries to evaluate the enabling condition for its execution. Because of the *and* relation between the value concepts, the simulator should be unable to evaluate the inner graph. Hence, the simulator should quit simulation, saying that the unsuitable concept types were present.

Simulator Output:

```
Thu Apr 26 19:48:31 EDT 2001
File simulated: badoperand.cgf

Graph has 4 concepts and 4 relations.
          adding a state concept with ident 5
          adding a state concept with ident 7
          concept 3 is a context
          concept 0 is a context
          concept 1 is a context
```

```
Initial Incident vector of size 1 received.

TIME: [0]
      STEP: [0]
      Found 0 value incidents.
      Found 0 state incidents.
      Found 0 event incidents.
      Found 1 action incidents.
            For action incident 1
            action is being initiated
      STEP: [1]
      Found 0 value incidents.
      Found 1 state incidents.
            For state incident 1
            ACTION: sleep (mode = active)
                  processing an AND or OR relation in evaluate
                  processing an EQ or NE relation in evaluate
                  The first concept is an action concept.
                  Its mode is: active
                  The second concept is a state concept.
                  Its state is: active
                  The final result is: true
                  First value is: 'true'
                  processing an AND or OR relation in evaluate
ERROR : Procedure 'evaluate' has unsuitable concept type(s) 'value' as the argument of the relation 'and'. SIMULATION ABORTED!
ERROR : Procedure 'sleep' has unsuitable concept type(s) 'state' as the argument of the relation 'enable_if'. SIMULATION ABORTED!

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Test Case 22 <loadandmultiplyreg.cgf>

Objective: This test case shows the output of the simulator for the *multiply* action, when the action involves device concepts that have not been prepared. (Changes to procedure prepare() in SimulationEngine.java; leave registers unprepared for simulation)

```
[Proposition:''
[action:*a'write']
[device:*b'register_reg1']
[device:*c'register_reg2']
[value:*d'v_15']
[value:*e'v_17']
[value:*f'v_23']
[device:*g'register_reg3']
[action:*h'write']
[action:*i'multiply']
(source?a?e)
(destination?a?b)
(delay?a?d)
(initiate_t?a?h)
(source?h?f)
(destination?h?c)
(delay?h?d)
(initiate_t?h?i)
(operand1?i?b)
(operand2?i?c)
(result?i?g)
(delay?i?d)]
```

Expected Output: The simulator should first initiate the *multiply* action. The action tries to read from device concepts that have not been prepared. This is seen as a missing relation, as the

device concept should be related to a value concept using the *contain* relation or the *containAt* relation.

Simulator Output:

```
Thu Apr 26 19:50:18 EDT 2001
File simulated: loadandmultiplyreg.cgf

Graph has 9 concepts and 12 relations.

TIME: [0]
      STEP: [0]
      STEP: [1]
            ACTION: write (mode = active)
ERROR : Procedure 'write', missing relation 'contain'. SIMULATION
ABORTED!

FILE SIMULATION DONE!
```

Conclusion: The simulator output matches the expected result.

****

Vita

Kiran S. Sastry

Kiran S. Sastry was born on 29th October 1974, in Bangalore, India. He completed his Bachelor of Engineering in 1997, from Sri Jayachamarajendra College of Engineering, affiliated to the University of Mysore, Mysore, India.

He was employed as a software engineer with Robert Bosch India Limited, Bangalore, India, for a period of around two years. During this period, his responsibilities included the design, implementation, testing and documentation of telecom software.

He joined Virginia Tech in the fall of 1999 to pursue his graduate studies in computer engineering. He graduated in spring 2001. He is currently working with Hughes Network Systems in their hardware division.