Visualization Feedback from Informal Specifications

by

Aniruddha Thakar

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

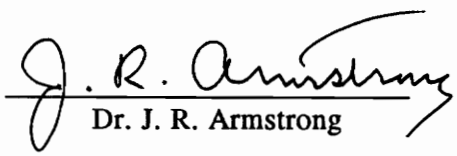
Master of Science

in

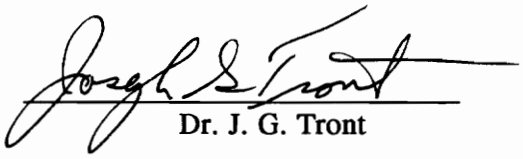
Electrical Engineering

APPROVED:

Dr. W. R. Cyre, Chairman



Dr. J. R. Armstrong



Dr. J. G. Tront

July, 1993

Blacksburg, Virginia

Visualization Feedback from Informal Specifications

by

Aniruddha Thakar

Dr. W. R. Cyre, Chairman

Electrical Engineering

(ABSTRACT)

This thesis describes the design and implementation of a system called the Model Generator that graphically models digital system specifications expressed in English. This research is part of the ASPIN project which has a long-term goal of providing an automated system for digital system synthesis from informal specifications.

Because of the versatility of the English language there can be more than one interpretation of specification sentences. So before these specifications are synthesized into formal models, it is necessary to obtain validation of their interpretation from the specification author. The specification sentences are mapped into a common knowledge representation, called *conceptual graphs*, by first parsing and then semantically analyzing them. The Model Generator then uses the conceptual graphs to generate a graphical model representing the meaning of the English specification sentence. This is done in two stages. First, the commands for drawing the icons used for the graphical representation are assembled by consulting an Interpretation Library and a conceptual type hierarchy. In the second stage, the icons used in the representation are displayed using an X-Windows interface. The Model Generator has been implemented in the C programming language under the X-Windows environment.

Acknowledgments

I would like to thank my advisor Dr. Walling Cyre for the challenging environment he made available, and the support and guidance throughout this research. I would like to thank Dr. J. R. Armstrong and Dr. J. G. Tront for serving as members of my committee and for their valuable suggestions.

Most of all, I would like to thank my wife, Sarita, for her love and support through all my endeavors.

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

1.1. Motivation

With the current rapid advancements in technology, the capabilities of VLSI chips and digital systems are increasing rapidly. This makes current digital designs obsolete in just a few years. Hence the issue before designers is not only to design systems that use the latest technology, but also to reduce design cycle times. Rapid and correct interpretation of design requirements is an important step in the development of new products. During the conceptual phase of product development, the design specifications are generally expressed in a natural language narrative supplemented by block diagrams, flow charts, and timing diagrams. As system development progresses, additional notations like schematic diagrams and wiring lists are introduced. Maintaining consistency, correctness, and completeness of the system specifications gets increasingly difficult. Design automation tools to extract, integrate and check the knowledge represented through the various notations are necessary if specification development times are to be reduced. After the knowledge has been extracted and integrated, it has to be translated into formal notations which can be used by design engineers.

Of all the various design notations, natural language is the least formal, but most versatile for specifying characteristics and behavior of systems. Unfortunately, because of

the richness of natural language, ambiguity can be introduced. There can exist multiple interpretations for a given set of specifications. Due to this, it becomes necessary that the interpretations of the natural language specification be validated before mapping to formal models. Thus it is desirable to develop a system that can provide a correct and easily understandable interpretation of the English specifications and obtain validation from the author before mapping to formal models.

Since such a system attempts to reduce design cycle time at the conceptual phase of product development, it has been implicitly assumed that the specification author is not adept in formal modeling and is untrained in the use of modeling tools such as hardware description languages (HDL). This means that a HDL cannot be used as feedback instrument. Clearly, the feedback should be easily understood by the user. After the English sentence is translated to a machine understandable form, the main task at hand is to translate this form to an easily understandable and correct representation. Diagrammatic (or iconic) representations are well suited to this task. This forms the basis for the development of the system, called the Model Generator, presented here. The Model Generator attempts to provide a visual feedback in a system that synthesizes the various forms of specification notations to formal HDL models. An overview of this system, called the Automated Specification Interpreterm, is presented in the next section.

A specification for a system can be both behavioral and structural. The structural information includes the hierarchical decomposition of the system, whereas the behavioral information indicates how the system operates. Block diagrams have been widely used as a visual tool to specify the structural aspects of digital systems. One drawback is that they do not represent behavior well. Petri nets model systems that exhibit asynchronous and

concurrent activity. The Model Generator uses block diagrams and Petri net notations to graphically model the two aspects of digital system specifications.

1.2. ASPIN System

The Automated Specification Interpreter (ASPIN) is a tool, currently uder development, that takes the various forms of informal specifications and creates formal engineering models from them [1, 2, 3]. The VHSIC hardware description language (VHDL) is the target formal language for the system. ASPIN uses the Model Generator to provide visual models of the input informal specifications to obtain author validation before generating VHDL behavioral models.

Figure 1.1 shows an overview of the ASPIN system objective. The various forms of design information are entered into the system and mapped into a common knowledge representation called *conceptual graphs*.

Natural language input is first parsed by the Parser. The Parser derives parse trees to represent sentence structure. Parse trees are used as input to the Semantic Analyzer. The Semantic Analyzer traverses the parse trees and consults a dictionary of canonical conceptual graphs to assemble conceptual graphs. These conceptual graphs are then used by the Model Generator for generation of graphical models. The Model Generator consults an Interpretation Library to generate the graphical model from the input conceptual graph. The model is fed back to the author for validation. The validated conceptual graph can then be processed and used to synthesize VHDL models simulating the behavior and characteristics of the system.

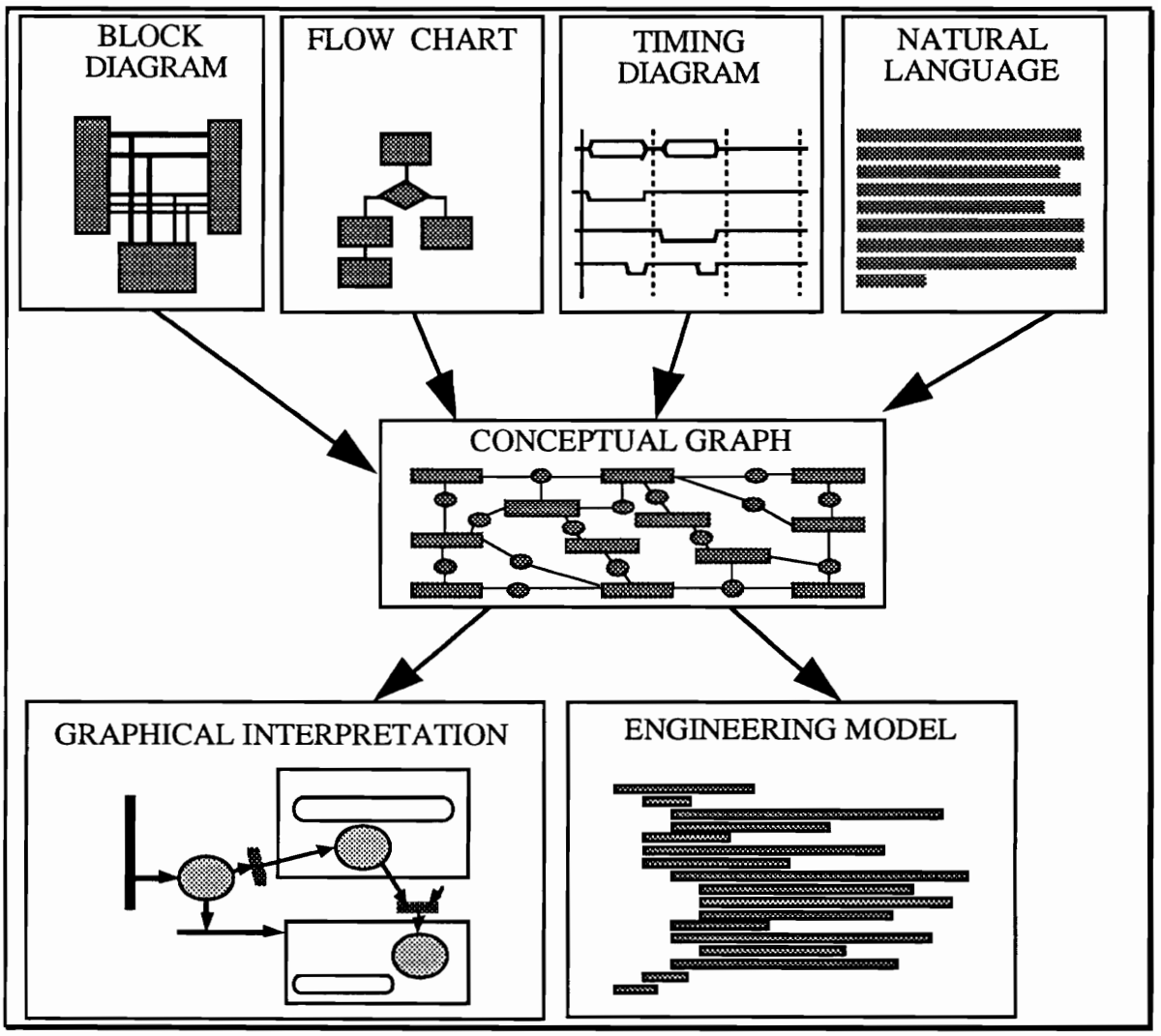


Figure 1.1. Overview of the ASPIN System Objective.

Synthesis of conceptual graphs to VHDL models is done using a program, called the Linker, that transforms them into a form that is used by the Modeler's Assistant [16]. The block diagram of the ASPIN system is presented in Figure 1.2.

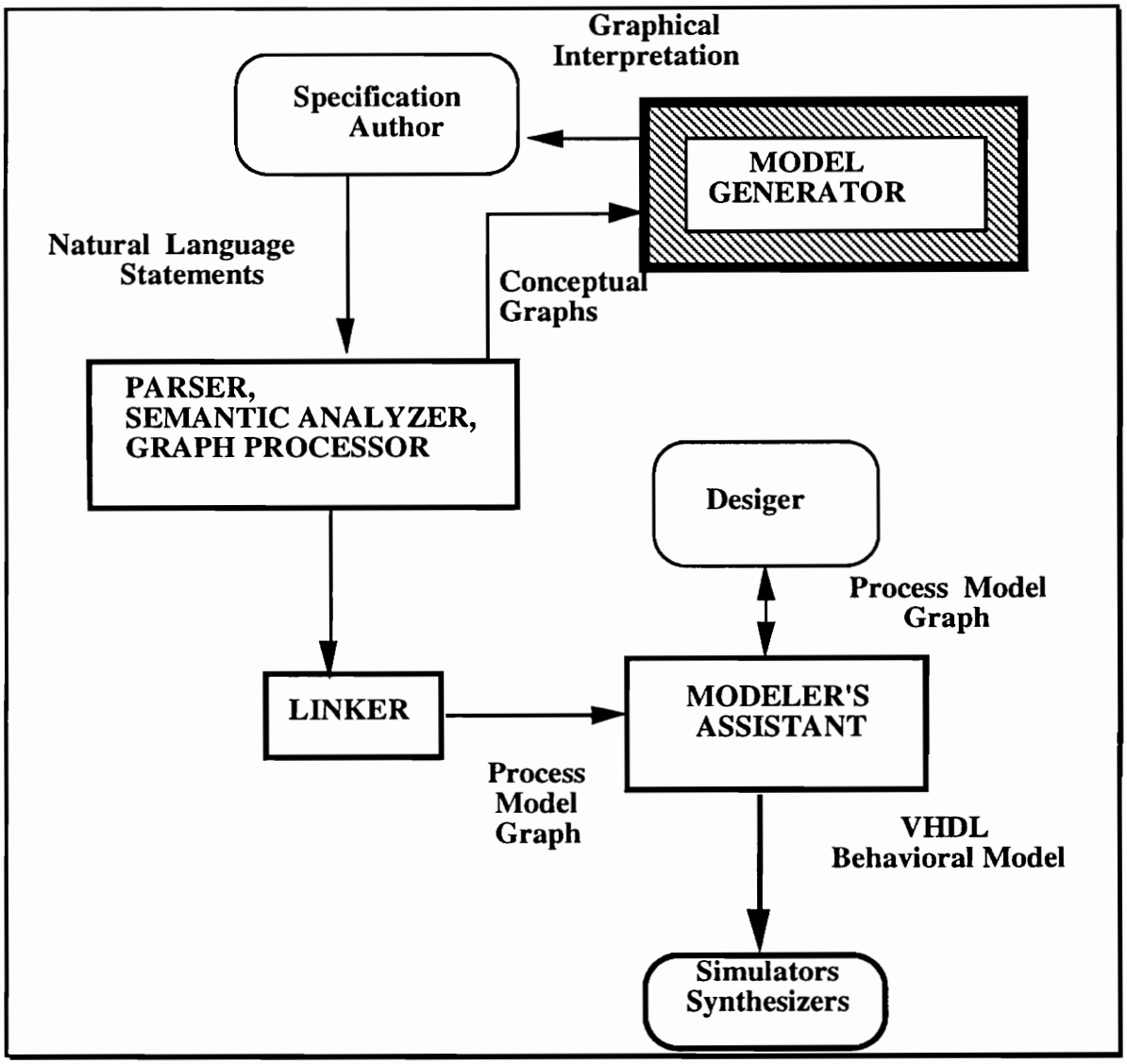


Figure 1.2. Block Diagram of the ASPIN System.

1.3. Overview of the Model Generator

The input to the Model Generator is a conceptual graph, produced by the Semantic Analyzer from English sentences. The Generator assembles an interpretation of the input sentence based on this conceptual graph.

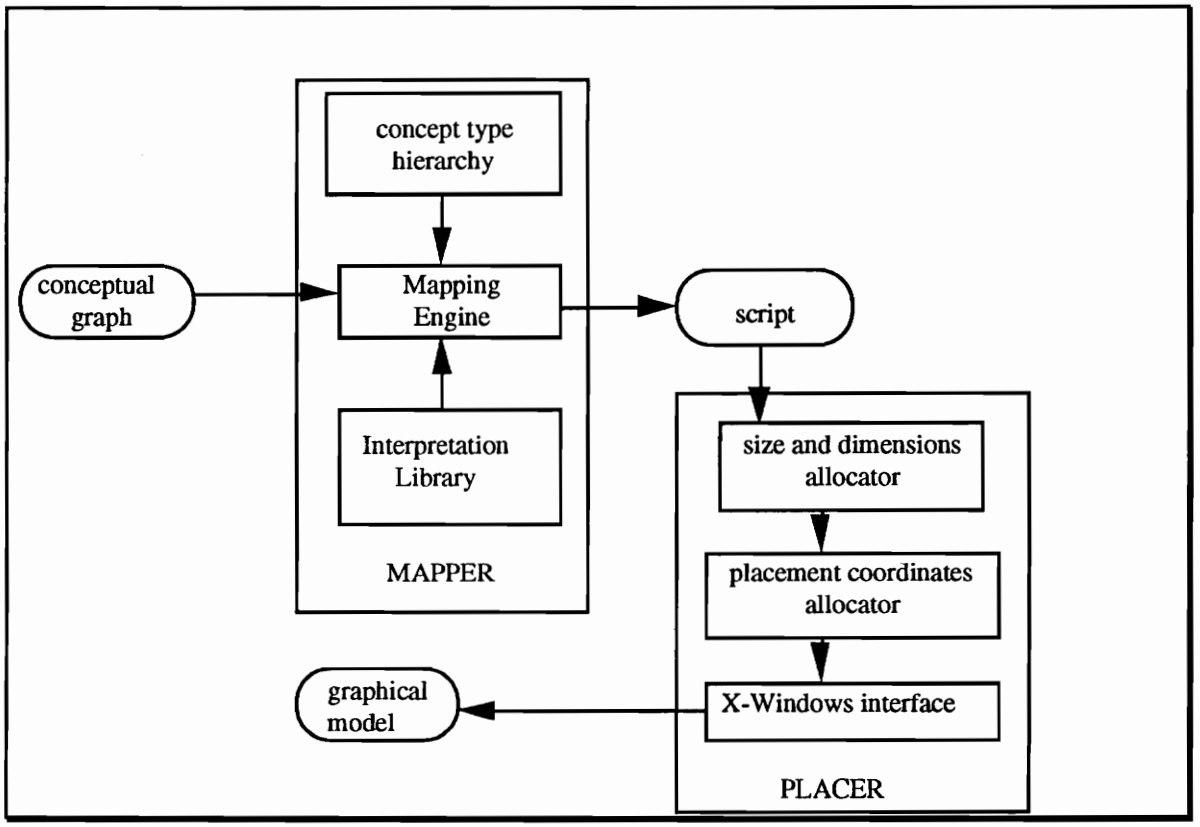


Figure 1.3. Block Diagram of the Model Generator

Figure 1.3 shows the block diagram of the Model Generator. It retrieves the set of graphical interpretations associated with the canonical conceptual graphs for the concepts in the input conceptual graph from a knowledge base (Interpretation Library). These interpretations are then instantiated. The graphical elements are assembled together to generate the complete graphical interpretation of the sentence. During the process of assembling, it is ensured that all redundancies and multiple references are removed. The generated graphical representation is then displayed to the user for validation.

There are two main modules - the Mapping Engine and the Placer. The input to the Mapping Engine is the input conceptual graph. Retrieval and assembly of the graphical

elements is done by the Mapping Engine. For each concept in the conceptual graph, its graphical representation is retrieved from the Interpretation Library. The system type hierarchy is also consulted to obtain the place of the concept in it. This information is used during the process of assembly. The graphical interpretations are in the form of drawing commands for the icons used for the representation. As each graphical representation is retrieved, it is instantiated using the information in the input conceptual graph. Then this instantiated picture is merged in the previously assembled picture. This involves joining of the two pictures and removal of redundancies. By continuing this process for all the concepts in the conceptual graph, the graphical interpretation for the entire sentences is obtained. The picture is in the form of drawing commands. This set of drawing commands is then supplied by the Placer.

The Placer takes the script and generates window coordinates as well as dimensions for the icons. First, sizing of each icon is performed. This is a recursive process involving sizing of outermost icons last. The coordinate generation for the icons is done using a force directed placement technique. This involves modeling the outermost icons as rings and all connections in the picture as springs. This physical system is placed in an initial configuration, which uses default coordinate placement. The system searches for an equilibrium configuration, which provides the final placement coordinates for the outermost icons. The inner icons have placement coordinates relative to outer icons. The Placer uses an X-Window interface to display the graphical representation. The interface uses X-Library (Xlib) subroutines for display and also provides user interaction.

Both the Mapper and Placer modules have been written in the C programming language under the X-Windows environment. The Model Generator currently runs on a Sun SPARCstation II platform.

1.4. Thesis Organization

This thesis is presented in seven chapters. Chapter 1 presents the motivation for this work and also provides the overview of the system. Chapter 2 reviews the related work done by other researchers. The topics include semantic analysis, conceptual graphs, Petri nets, automatic graph drawing and operational definitions for system requirements. The details of conceptual graphs, block diagrams and Petri nets are presented in Chapter 3. Also presented here is the semantic model of the system along with the representation scheme used. Chapter 4 provides the details of the Mapping Engine. The mapping process is explained in detail. Also, the flowchart for the mapping algorithm is presented. The details of the placement process are presented in Chapter 5. Also presented in this chapter is the force-directed placement technique and the X-Window interface of the system. Chapter 6 contains model generation examples. An illustrative example is described in detail. Other examples are also presented in brief. Chapter 7 suggests some directions for future work.

The report has four appendices. Appendix A contains the Users Manual of the Model Generator. This also contains the file structure of the system. Appendix B contains a list of the vocabulary currently used by the generator. Appendix C provides the Interpretation Library. A suite of test sentences and their graphical interpretations is provided in Appendix D.

Chapter 2. Related Research

2.1. Conceptual Graphs

Conceptual graphs were developed by John Sowa. These were based on existential graphs developed by Charles Pierce. Existential graphs were a graphical form of formal logic. Conceptual graphs can represent semantic information as well as assertions about events. They have been defined as labeled, bipartite and directed graphs consisting of concept nodes and conceptual relation nodes. Further details are presented in Section 3.1. John Sowa outlined four basic operations for conceptual graphs. These are *copy, restrict, join* and *simplify*. These operations can be used to generate new conceptual graphs from existing ones. A more rigorous and formal discussion of conceptual graphs can be found in John Sowa's book [17].

Conceptual graphs were used by John Sowa and Eileen Way to implement a semantic interpreter [18]. They started with parse trees generated by a parser that show the syntactic structure of the sentence. Their system generated conceptual graphs that represent the meaning of the sentence. The interpreter was written in the Programming Language for Natural Language Processing (PLNLP). The interpreter generates conceptual graphs by joining canonical graphs associated with each word of the input. The result is a larger

graph that represents the entire sentence. The parse trees were used to guide the joining process. The operations used are implemented as PLNLP subroutines in their interpreter.

The conceptual graph model was also used by Jean Fargues, et. al., in their semantic interpreter which was written in PROLOG [6]. In this paper, the authors discuss the representational and algorithmic power of the conceptual graph model for natural language semantics and knowledge processing. The ability to generate new conceptual graphs using the four formation rules is also illustrated by the authors. They also describe their interpreter in which the terms are conceptual graphs and the unification algorithms have been specialized. A resolution method that allows expression of large amount of background knowledge in terms of conceptual graphs is central to their interpreter. This work makes clear the ability of the conceptual graph model to represent knowledge and its utility in performing deductions on large linguistic and semantic domains.

The semantic analyzer used by the ASPIN system generates conceptual graphs from parse trees of English sentences [8]. The conceptual graphs are built by joining *canonical* conceptual graphs that represent words found in the input sentence. A canonical graph for a word is defined as a graph that "represents the default ways that concepts and relations are linked together in well-formed sentences". The parse trees direct the semantic analysis, which relies on a set of rules based on the grammatical structures found in the input sentence. The semantic analyzer is written in Quintus PROLOG.

The works reviewed above illustrate the usefulness of conceptual graphs in representing the knowledge present in English sentences. Also, references 6 and 8 present methods for generating larger conceptual graphs from canonical conceptual graphs for individual concepts. The mapping process in the Model Generator involves assembling

graphical representation of canonical graphs for the concepts in the input conceptual graph (instead of words from a parse tree as done above), to obtain the graphical interpretation of the complete sentence. The details of the formation technique used by the Mapping Engine can be found in Chapter 4.

2.2. Operational Definitions for System Requirements

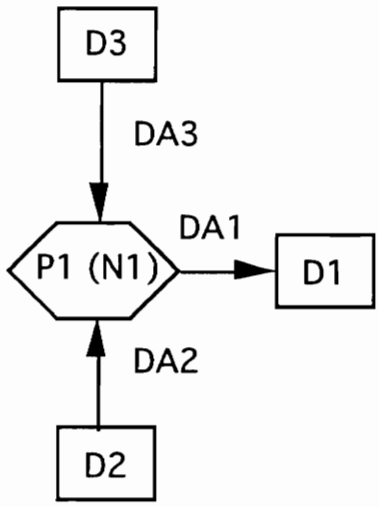
To achieve automatic mapping between requirements and design, the semantics of each construct in the requirement model have to be formally defined. These definitions constitute the core of a knowledge base in an automatic design systhesis tool.

Kar-Wing Edward Lor introduced in his paper formal definitions to system requirements as the basis of design automation [10]. The focus of this paper is on the mapping of requirements expressed in *stimulus-and-response* model to a design expressed in the UCLA *graph model of behavior* (GMB). The stimulus-and-response model is a graph-based model in the form of *system verification diagram* (SVD). The primary purpose of this model is to demonstrate a static event-dependency relationship among modules or subsystems. A collection of SVDs, each representing a subsystem, constitutes the *operations concept* of the whole system. An SVD is simply a directed graph, in which each node corresponds to the requirement specification of a component.

The paper presents a scheme to formally define the constructs of a requirement language. The formal definitions of the requirement constructs provide the foundation of an experiment in design automation. On top of SARA (System ARchitect Apprentice) design environment, developed in UCLA, there is a design synthesizer that automatically generates GMB skeletons from the requirements. SARA is a requirement-oriented method for designing concurrent systems. Two SARA models represent a design. The *structural model* (SM) represents the hierarchical decomposition of the system, whereas the GMB shows how it operates.

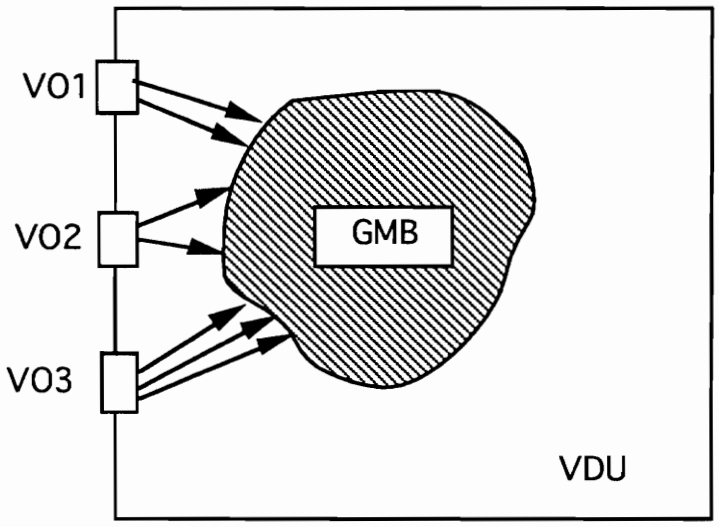
The structural model has three primitives - *module*, *socket* and *interconnection*. A module represents a system component and hierarchical decomposition is attained by refining a module into sub modules. A module is connected to another module by an interconnection bridging two sockets, the modules' communication ports. The GMB models three different but related aspects of the system - control, data and interpretation. Control domain of GMB describes concurrency, synchronization and precedence relations in a graph. The control graph is a directed hypergraph. The GMB data graph is bipartite data graph. Provided in the data domain is the system's data storage units, their values at various system states and their access rights by the data processing units. The interpretation domain defines simulation timing control, data transformations of the data processor and control decisions of the associated control node. This domain can use any programming language. The author indicates that formal definition of the requirement constructs can be undertaken using Petri net formalisms instead of the GMB.

An example of the GMB is shown below. The *controllled data processor* P1 is initiated by *control node* N1.



P1 reads data from *datasets* D2 and D3, via *data arcs* DA2 and DA3, respectively, and writes the result into the dataset D1 through arc DA1. Controlled data processor, control node, dataset, and data arc are some of the GMB primitives.

The SM can be illustrated by the operational definition of a video display unit (VDU) presented in the paper. The GMB for the VDU resides in the SM and is not reproduced in the Figure below due to its complexity. The SM module called the *VDU* has three sockets, each of which can be used to communicate with some other subsystem related to the VDU. VO1, VO2, and VO3 are the sockets and the SM is represented by the outer box.



To establish a mapping between requirements expressed in the stimulus-and-response model and design, the author uses two SARA models as the target notation. Use of the two models, GMB for modeling behavior, and SM for modeling the structure of the system, indicates that specifications consist of behavioral and structural aspects of the system. The primitives of the SM correspond to some of the ASPIN semantic types. For example, the *module* primitive of SM is similar to the DEVICE concept used by ASPIN.

Also, the *socket* and *interconnection* primitives correspond to the ASPIN semantic types PORT and CARRIER respectively. The *dataset* primitive of the GMB is similar to the VALUE concept used in ASPIN. Since the primary intent of the Model Generator is a prompt and readable feedback, instead of a formal notations like SM and GMB, use of directed general graphs appears to be more suitable. Modeling natural language specifications is undertaken using the Petri net graph notation (akin to GMB) and block diagrams (instead of SM).

2.3. Petri Nets

Petri nets are used as a basic model for systems of asynchronous concurrent computation. The theory of Petri nets has developed from the work of Carl Adam Petri [22]. James Peterson in his paper surveys the basic concepts and uses of Petri nets [11]. Some of the important topics presented in this paper are structure of Petri nets, their properties, and research into the analysis of Petri nets.

A Petri net is an abstract, formal model of information flow. Petri nets are useful in modeling systems that exhibit asynchronous and concurrent activities. The pictorial representation of Petri nets, called *Petri net graphs* are useful in representing Petri net operations. The Petri net graph models the static properties of a system. The graph contains two types of nodes: circles (called *places*) and bars (called *transitions*). These nodes are connected by directed arcs from places to transitions and from transitions to places. A Petri net graph is a directed, bipartite graph. In addition to its static properties represented by the graph, a Petri net has dynamic properties that result from its 'execution'. The execution of Petri nets is controlled by the position and movements of markers (called *tokens*) in the Petri net. A Petri net with tokens is called a *marked* Petri net.

Petri nets are useful in modeling *concurrency* and *parallelism*. In the Petri net model there is no need to synchronize the actions. When synchronization is required, Petri nets can easily model that situation. Another major feature of Petri nets is their *asynchronous* nature. There is no inherent measure of time or flow of time. Events take variable amounts of time in real life; the Petri net model reflects this variability by not depending upon a notion of time to control the sequence of events. A Petri net, like the

system that it models, is viewed as a sequence of discrete events whose order of occurrence is one of possibly many allowed by the basic structure. The intent of the model can be indicated to the human observer by means of *labels*. These labels do not, in any way affect the execution of the net. Another valuable feature of Petri nets is their ability to model a system *hierarchically*. An entire net may be replaced by a single place or transition for modeling at a more abstract level (*abstraction*) or places and transitions may be replaced by subnets to provide more detailed modeling (*refinement*).

Since the Model Generator attempts to graphically model system specifications, the Petri net graph notation can be utilized for representation. The ability to label Petri net graphs is useful to indicate to the user the information which, even though not critical to the execution of the system, may be of interest. Clearly, the static properties specified by requirement specification can be modeled well by the Petri net graph notations. The Model Generator relies extensively on this notation to graphically model behavior. But, unlike in Petri net graphs, which use only circles and transition bars, the Model Generator employs various other icons to represent different concepts in the specification. This increases the readability of the graphical feedback.

2.4. Automatic Graph Drawing

The main quality desired from graphical representations is 'readability'. Diagrams provide the user with a freindly interface to validate representations and direct queries. Automatic graph drawing is particularly desirable because it integrates the phases of conception and production of diagrams and decreases the time required to generate diagrams. Force-directed placement for automatic generation of graphs is a well researched technique, and is used by many graph drawing and placement algorithms [5, 7, 13, 14].

Neil Quinn proposed a procedure for placing electronic components on a printed circuit board using classical laws of Physics [13]. The procedure is essentially force directed, solving a set of simultaneous non-linear differential eaquations which are derived from the laws of Physics. This work along with his paper with M. Breur [14] formed the basis for later work on drawing undirected graphs using force directed techniques. Peter Eades presented a method for graph drawing using a spring-embedder model [5]. To layout a graph, vertices are replaced by steel rings and edges with springs to form a mechanical system. The vertices are placed in some initial layout and let go so that the spring forces on the rings move the system to a minimal energy state. The positions of the vertices are obtained in this stable state.

Eades used the following two equations for attractive and repulsive forces repectively,

$$F_a = C1 \log (d/C2) \text{ and}$$

$$F_r = C3/\text{ sqr}(d),$$

where **d** is the length of the spring and **C1**, **C2** and **C3** are constants.

Thomas Fruchterman and Edward Reingold based their work on the earlier work done by Eades and improved his basic algorithm to reduce time and computational complexity [7]. They replaced the logarithmic springs used by the Eades and experimented before finalizing on the following two force equations,

$$F_a(d) = d^2/k \quad \text{and}$$
$$F_r(d) = -k^2/d,$$

where **d** is the distance between the vertices and **k** is the optimal distance between the vertices.

The algorithm used by the Placer in the Model Generator is an implementation of the algorithm proposed by Fruchterman and Reingold. The outermost icons in the graphical representation of the sentence are modeled as rings and all the connectives in the picture as springs. The initial configuration is defined by the default display settings in the Interpretation Library.

Chapter 3. Fundamentals and Representation Scheme

3.1. Conceptual Graphs

The earliest forms of conceptual graphs, called existential graphs, were developed by Charles Peirce in the nineteenth century . Existential graphs served as a graphical notation for symbolic logic. The conceptual graphs used by the Model Generator were developed by John Sowa [12]. Conceptual graphs can represent semantic and episodic information. Semantic information consists of word definitions, constraints on the word usage in well-formed sentences and information about defaults. Episodic information consists of assertions about particular events and things. Conceptual graphs are used as the common knowledge representation in the ASPIN system because they map directly into formal logic, which can be used for reasoning and consistency checking.

A conceptual graph is defined as a finite, labeled, bipartite, directed graph, consisting of concept nodes and conceptual relation nodes. A *concept* node is an ordered pair consisting of a concept type label, and a referent label. It can represent an object, action or feature. A partial ordering is defined over the concept types. This partial ordering forms the *concept type hierarchy*. A *relation* node is a pair consisting of a

conceptual relation type label and a referent label, and a partial ordering may be defined over the set of conceptual relation types. Relation nodes identify relationships betwen concepts. An arc in a conceptual graph links a concept to a conceptual relation or a conceptual relation to a concept.

Conceptual graphs can be represented graphically as well as in linear text. In the graphical representation, the concepts are drawn as boxes and conceptual relations are drawn as ellipses. The arcs linking the concepts and relations are drawn as arrows. In the textual representation of conceptual graphs, concept nodes are contained in square brackets. Each concept also has a unique concept identifier preceding its type field. The relation nodes are contained in parentheses. Arrows represent the arcs in the graph.

In Figures 3.1 and 3.2 below both the representations of conceptual graphs are shown. The two graphs are read "the *relation* of *concept 1* is *concept 2* ".

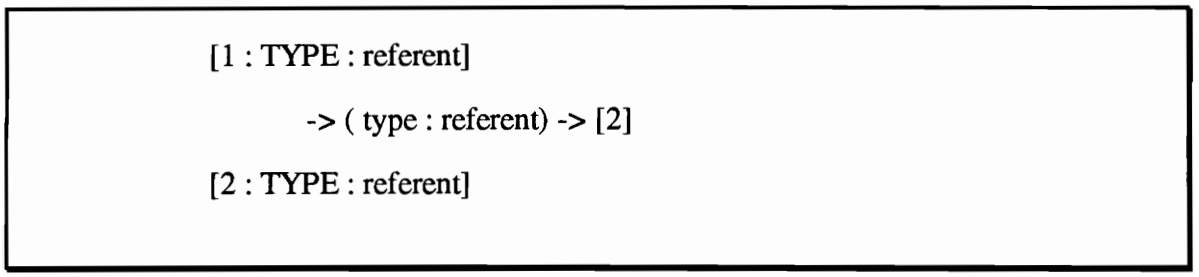


Figure 3.1. Conceptual Graph (Linear Textual Form).

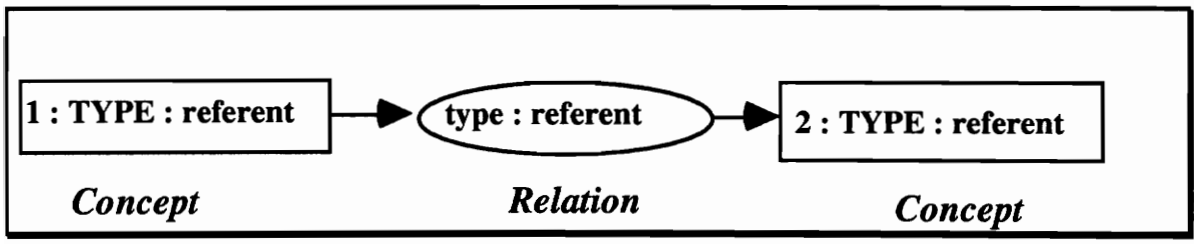


Figure 3.2. Conceptual Graph (Pictorial Form).

There can be several different types of markers in the referent field. The *generic* marker is denoted by an asterisk '*' in the referent field of a concept, and it represents an unspecified individual of the given type. An *individual* marker is denoted by the pound sign '#' followed by an identification number. This identifies a particular instance of the concept type. Names of individual referents can also form labels in the referent field.

Figure 3.3 shows the conceptual graph for a sentence in the graphical form. In this conceptual graph there are four concepts and three conceptual relations. The 'head' or the root concept is the concept READ. The agent of the action is PROCESSOR, a subtype of the basic semantic type DEVICE. The relation is identified by the relation node *agnt* . The object, identified by relation node *obj*, for the action is DATA which is a subtype of VALUE. MEMORY, another subtype of DEVICE, is the source for the data. Words "processor", "data", and "memory" are the names of the concept types and are placed in the referent field.

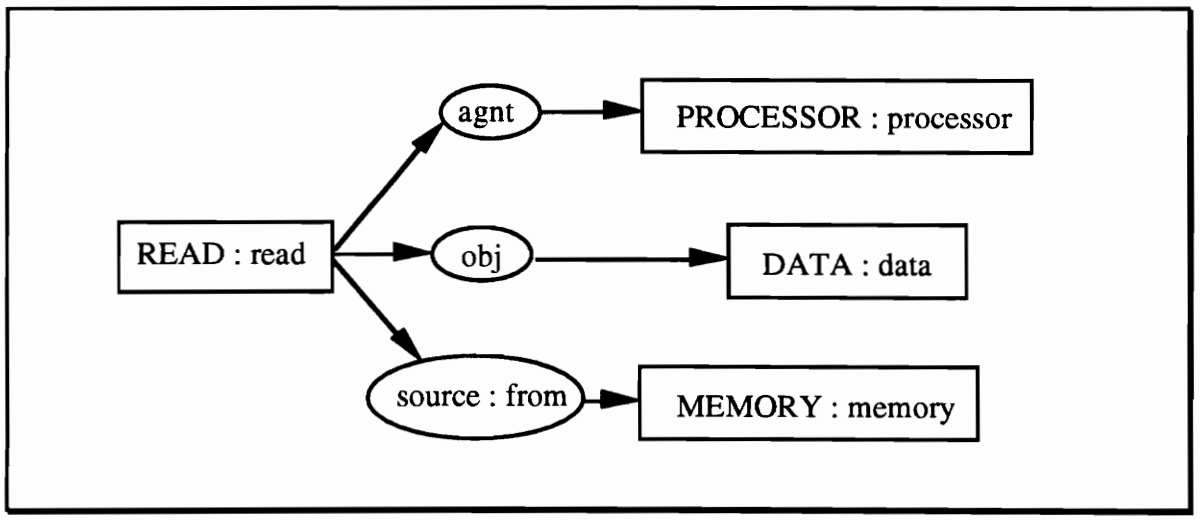


Figure 3.3. Conceptual Graph for "*The processor reads the data from memory*"

The linear form for the above conceptual graph is shown below.

```
[ 1 : READ : read ]
        -> (agnt) -> [ 2 ]
        -> (obj) -> [ 3 ]
        -> (source : from) -> [ 4 ]
[ 2 : PROCESSOR : processor ]
[ 3 : DATA : data ]
[ 4 : MEMORY : memory ]
```

Figure 3.4. Conceptual Graph in Linear form.

3.2. Semantic Model

The set of relationships, that exist between concepts, as well as the knowledge needed to understand these relationships is contained in ASPIN's semantic model. The main part of the semantic model is the set of canonical conceptual graphs, the conceptual type hierarchy, and the set of conceptual relations linking the concepts. Digital specifications written in English use a subset of the English language. A technical sublanguage grammar and vocabulary can be defined for this subset [4]. Each concept within the sublanguage used by the ASPIN system is classified into the conceptual type hierarchy.

The basic semantic types used in the system are: *DEVICE, VALUE, ACTION, EVENT, STATE, RELATIONSHIP and UNIT.* The DEVICE concepts of a system comprises its hardware elements. The software and data of a system are VALUE types. These are objects which control the activities of devices and may also be the subject of DEVICE operations. ACTIONS are the names of the operations and activities performed by the DEVICES. A simultaneous discontinuity in one or more actions is an EVENT. An ACTION is finished or started as a result of an EVENT. Relationships may be structural, behavioral or attributive. UNITS are basically measures and the units of interest are: memory and time. The various types and their subtypes constitute the concept type hierarchy of the Model Generator. The basic concept types with their definitions are tabulated in Table 1 [8].

Table 1. Basic Conceptual Types and their Definitions.

Conceptual Type	Definition
OBJECT	Any unit of information, software or hardware.
ACTION	An activity or process, usually having a duration.
STATE	Stative verbs and arithmetic/logical relations.
STRUCTURE	An attribute describing the organization of a structured concept.
EVENT	A discontinuity in one or more actions.
MEASURE	An attribute of the dimension or quantity of an object or action.

3.2.1. Concept Type Hierarchy

The conceptual type hierarchy is a partial ordering defined over a set of types based on levels of generality. The hierarchy is a lattice, therefore any two conceptual types classified in the hierarchy have a least common supertype as well as a greatest common subtype. At one end of the lattice is the universal type and at the other end is the absurd type. Each node in the lattice represents a concept type within the domain of digital system specification.

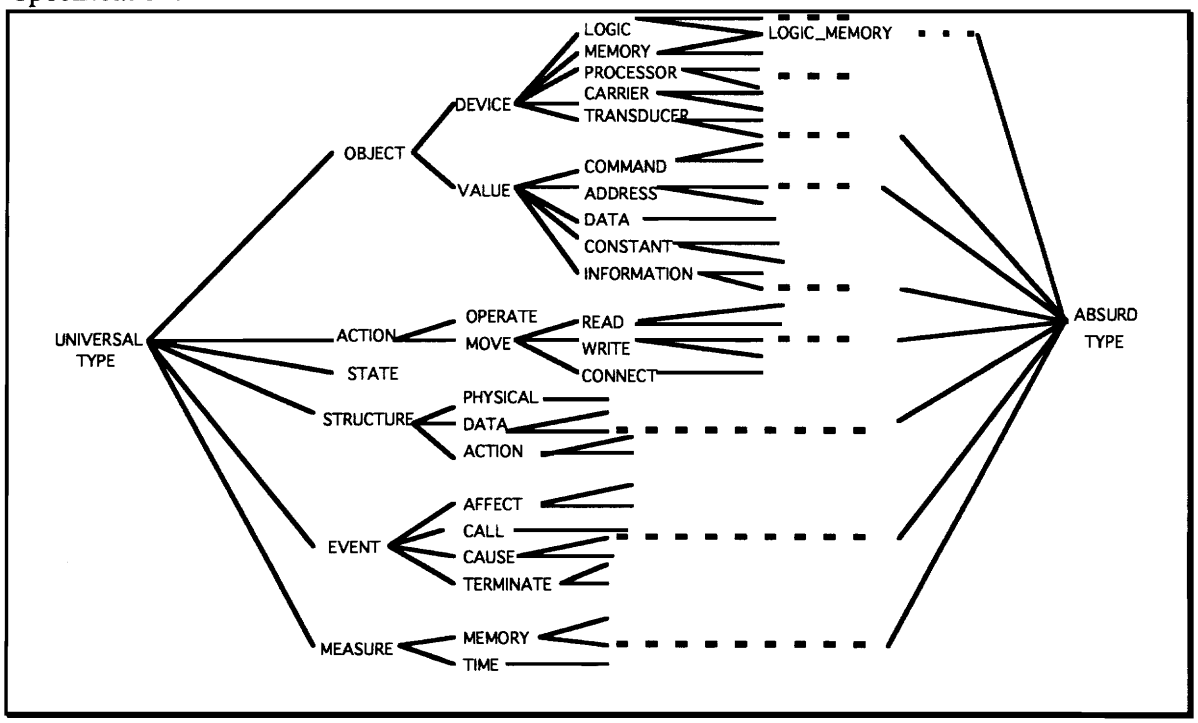


Figure 3.5. Partial Conceptual Type Hierarchy

Figure 3.5 shows the partial hierarchy used by the system. Every concept in the lattice is a subtype of the universal type. Similarly, no concept is a subtype of the absurd type. The more specific types are on the right. For example, *LOGIC* is a subtype of *DEVICE*, whereas *LOGIC_MEMORY* is a subtype of *LOGIC*. The concept types defined

in the lattice are like variables in a programming language. When a particular concept is used in a sentence, then it represents the name of the concept type defined in the lattice.

3.2.2. Conceptual Relations

The conceptual relations describe the way the concepts are related. They specify the roles that the concepts play with respect to each other.

Table 2. Conceptual Relations and their Definitions.

Relation (abbreviation)	Definition
accompaniment (**acc**)	a device or value that accompanies the object of an action.
after (**after**)	the action or event completes after the end of an interval or event.
agent (**agnt**)	the device that performs the action.
and (**and**)	conjunction relation indicating subparts.
attribute (**attr**)	a characteristic of the object.
before (**before**)	the action or event completes before the interval or event begins.
condition (**cond**)	the condition that enables the action or event.
destination (**dest**)	the destination object of an action.
determiner (**det**)	a word that specifies an individual referent.
frequency (**freq**)	the frequency at which the action occurs.
in (**in**)	indicates the state of containment of an object by a compatible object.
instrument (**inst**)	the device to implement the action or event.
location (**loc**)	location of a stored value.
modal (**mod**)	an auxiliary that modifies a verb.
name (**name**)	the name of an object.
negation (**neg**)	indicates action does not occur or condition does not exist.
object (**obj**)	the concept that is the object of an action or event.
operand (**opnd**)	a value used by the operation.
purpose (**purp**)	the purpose of device or action.
quantity (**quant**)	the number of objects.
result (**rslt**)	condition or value resulting from an action or an event.
size (**size**)	the size of an object.
source (**src**)	the source of an object of an action or an event.
type (**type**)	a description of the type of object.
value (**val**)	the value held by a device or a value.

Table 2 above provides the list of conceptual relations used by the Model Generator and also provides their definitions. The bold face strings in parenthesis are abbreviations for the conceptual relation types.

3.2.3. Canonical Conceptual Graphs

For every concept there exists one or more *canonical conceptual graphs* that specify the constraints by which attachments between the conceptual graph for that concept and other conceptual graphs can be made during semantic analysis to represent the meaning of sentences. A canonical conceptual graph is a conceptual graph that represents the meaning of a word.

Sowa has defined a set of formation rules that may be used to build new conceptual graphs from a set of existing conceptual graphs. The four formation rules are *copy*, *restrict*, *join* and *simplify*.

The *copy* rule states that an exact copy of a canonical graph is also canonical. *Restrict* replaces the type label of a concept with the label of a subtype. For example, the concept [PROCESSOR : M68HC11] can be restricted to [MICROCONTROLLER : M68HC11] because MICROCONTROLLER is a subtype of PROCESSOR. This rule can also replace a generic referent with an individual referent. The replacement is allowed only if the new referent conforms to the new type label. Thus the generic concept [PROCESSOR : *] can be restricted to [PROCESSOR : M68HC11] only if the new referent M68HC11 conforms to the type PROCESSOR.

Join creates a single graph by merging two graphs on a single matching concept. Matching concepts have identical types and referents. If one of the concepts is generic, it can be restricted to match the concept it is being merged with. When two concepts are merged, the relations list of one concept is added to the relations list of the other concept.

The former concept is then removed from the graph. Figure 3.6 shows the canonical graphs for INCREMENT and DATA.

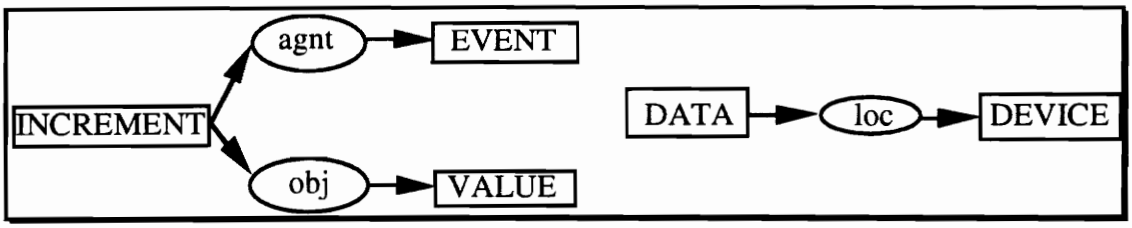


Figure 3.6. Canonical Graphs for INCREMENT and DATA.

These two graphs can be joined using first by restricting the graph for INCREMENT as shown below.

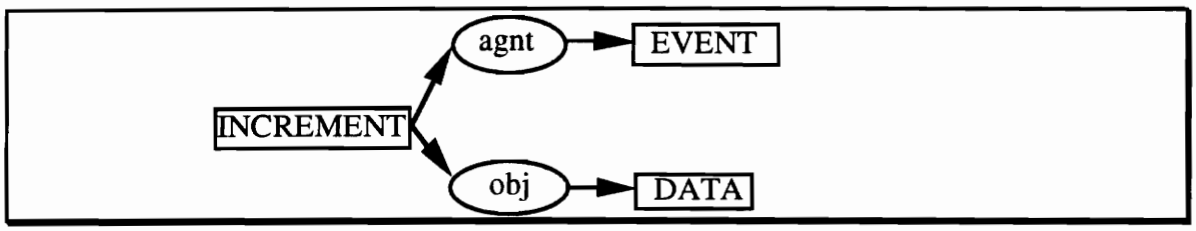


Figure 3.7. Canonical graph of INCREMENT after Restriction.

In Figure 3.7, the generic concept VALUE has been restricted to DATA. Figure 3.8 shows the resulting graph from the join of the two graphs in Figure 3.6.

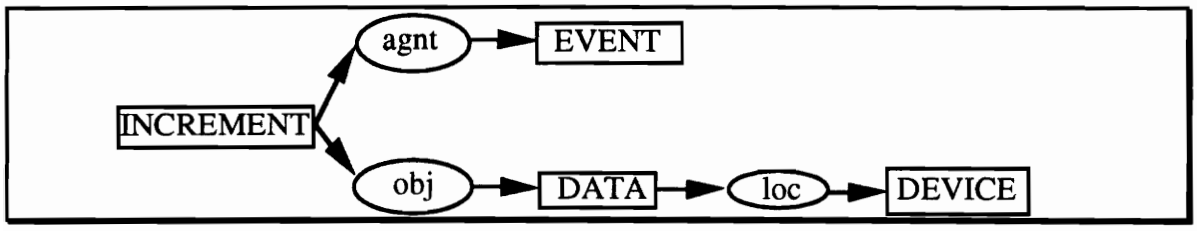


Figure 3.8. Join of Canonical Graphs.

Simplify deletes any duplicate relations resulting from joining graphs. Two relations are considered duplicates when they have the same type and referent and are linked to the same concepts in the same order.

Since words in English can be used in several senses, they often have more that one canonical graphs. A second canonical graph for INCREMENT is shown in Figure 3.9. The difference is that the type of concept that can be attached to the agent role is a DEVICE instead of an EVENT.

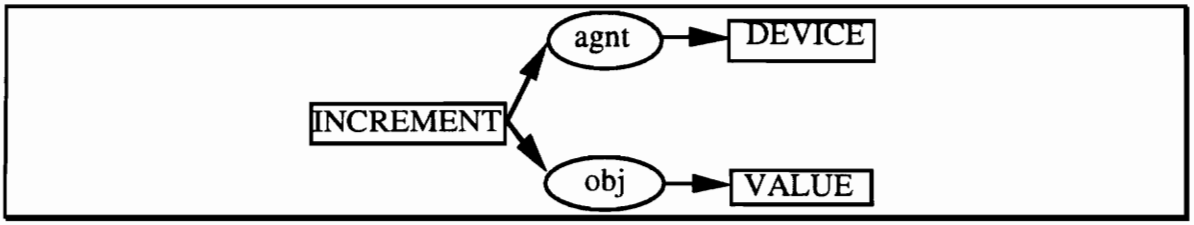


Figure 3.9. Alternative canonical graph for INCREMENT.

To differentiate between the different possible canonical graphs, they are provided with different index numbers. This is done by attaching a special relation called *gindx* (graph index) by the Semantic Analyzer. This aids the Model Generator in identifying the correct canonical graph for concepts in the input conceptual graph.

Included in the canonical graphs as referents of conceptual relations are *role markers*. They direct the Semantic Analyzer during the attachment of canonical graphs. They can either be *literal* or *structural*. Literal role markers correspond to actual words, such as prepositions and subordinate conjunctions, used in the input sentences. Examples of literal role markers are *in* and *into*. Structural role markers tell the Semantic Analyzer

how to link a word's syntactic use with its semantic use. If no role marker is specified, the referent field contains the string *null*.

3.3. Representation Scheme

The Model Generator uses block diagrams and Petri net notations to model the input specification sentence. Block diagrams and Petri net notations are presented below. Then the icons used in the representation scheme of the system are introduced.

3.3.1. Block Diagrams

A block diagram is a graphical tool that depicts interconnections and hierarchical nesting between physical units. Block diagrams have also been used as a mathematical modeling tool as opposed to being simply a visual aid for describing systems. Computer languages that can directly provide solutions to mathematical models by using their block diagram model have been proposed [15]. Various block diagram CAE tools are also available [19].

Block diagrams consist of two types of elements: *blocks* and *carriers*. A third element, *port*, can be defined as a point on the block at which an interconnection terminates. By using these three elements, a block diagram can be defined as a labeled, tripartite directed graph consisting of block nodes and carrier nodes which are connected by ports [3]. By nesting blocks inside other blocks hierarchical structure of systems can be

illustrated. For example the nested block diagram shown in Figure 3.10. shows that Memory A is decomposed into two smaller memories, A1 and A2.

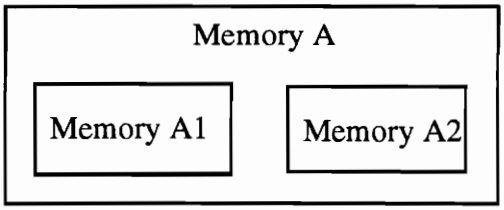


Figure 3.10. A Nested Block Diagram.

3.3.2. Petri Nets

Carl Adam Petri developed a model of information flow in systems [22]. The model was based on the concurrent and asynchronous operations by systems and the fact that relationships between parts of systems could be represented by a graph, or net. Researchers at Applied Data Research, Inc., provided the early theory, notations and representations of Petri nets [23]. They showed that Petri nets could be used in modeling and analysis of systems of concurrent processes. From this work, the use of Petri nets has spread widely.

Petri nets are composed of two basic components: a set of places $\boldsymbol{P}$ and a set of transitions $\boldsymbol{T}$. Two functions, $\boldsymbol{I}$, the *input* function, and $\boldsymbol{O}$ the *output* function, define the relationship between the places and transitions. The input function defines, for each transition t_j, the set of input places for the transition $\boldsymbol{I}(t_j)$. The output function defines, for each transition t_j, the set of output places for the transition $\boldsymbol{O}(t_j)$. Formally, a Petri net is defined as a four-tuple C= (P, T, I, O). In Petri net graphs, the input and output functions

are represented by directed arcs from places to transitions and from transitions to places. If the place is an input of the transition then the arc is directed from the place to the transition. Similarly, if an arc is directed from a transition to a place, then the place is an output of the transition.

For the Petri net structure given below, the corresponding Petri net graph is shown in Figure 3.11. The structure defined as a four-tuple is,

$$C = (P, T, I, O)$$

$$P = \{p_1, p_2, p_3\}$$

$$T = \{t_1\}$$

$$I(t_1) = \{p_1, p_2\} \qquad O(t_1) = \{p_3\}$$

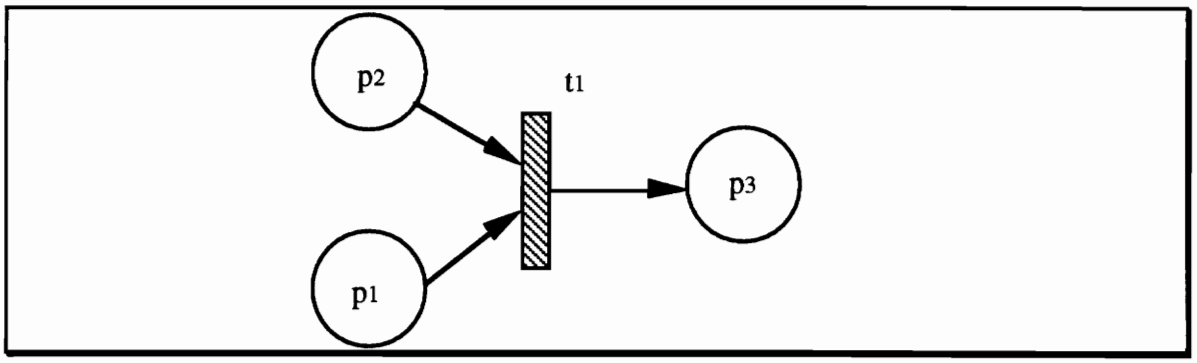


Figure 3.11. Petri Net Graph.

Since the arcs are directed, a Petri net graph is a *directed* graph. Also, since the nodes in a Petri net graph can be partitioned into two sets (places and transitions) such that each arc is directed from an element of one set to an element of the other set, a Petri net graph is *bipartite*.

A marking μ of a Petri net is an assignment of tokens to the places in that net. The vector $\mu = (\mu 1, \mu 2, \ldots, \mu n)$ gives for each place in the net (n in this case), the number of tokens in that place. On a Petri net graph, tokens are represented by small solid dots inside circles representing places of the net. For the Petri net structure presented above, if we define a marking $\mu = (1, 0, 1)$, the resulting *marked* Petri net $M = (P, T, I, O, \mu)$ can be represented as in Figure 3.12 .

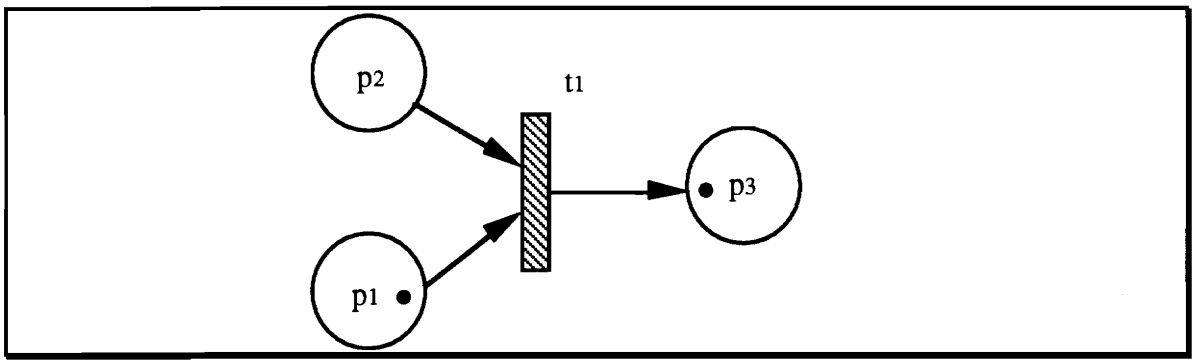


Figure 3.12. A Marked Petri Net Graph.

The movement of tokens allows Petri nets to model dynamic behavior of systems. The rules for token movements are:

1) tokens are moved by the *firing* of the transitions;

2) a transition must be *enabled* for it to fire;

3) a transition is enabled when all its input places have a token in them; and

4) the transition fires by removing the enabling tokens from their input places and generating new tokens which are deposited in the output places of the transition.

The Model Generator uses the Petri net graph notations to represent the static relationships and dependencies between actions and events. To improve readability of the graphical models, dotted arrows are used to indicate control dependencies to distinguish

them from data flows and carriers (solid arrows and lines). Also, *inhibtory* control of an ACTION by an EVENT is denoted by an arrow terminating in a bubble rather than an arrowhead. This is an extension to the basic theory of Petri nets [22].

Figure 3.13 shows the graphical model of "*action A enables action B*".

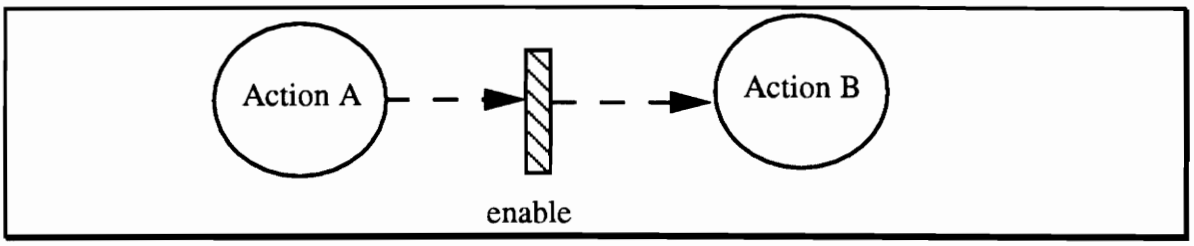


Figure 3.13. Graphical Model of " *action A enables action B* " .

The direction of the dotted arrow provides the source and destination of the control dependency between actions A and B.

Figure 3.14 shows the graphical model for " *action A disables action B*". The bubbled arrow indicates the inhibitory control of action A over action B.

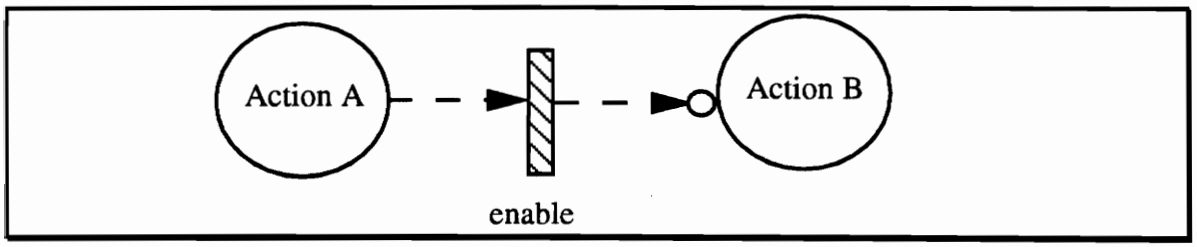


Figure 3.14. Graphical Model of " *action A disables actoin B* ".

Further details on the different icons used by the Model Generator are provided in the next Section.

3.3.3. Icons used in the Representation Scheme

The Model Generator represents the basic concept types with different shapes, or icons. These icons are assembled together to provide the graphical interpretation of the input specification.

Figure 3.15 shows the various shapes used to represent the basic concept types. The icons like rectangle, rounded rectangle, circle, transition bar and solid line represent the concept types. Connectives can either be solid arrows or dotted arrows.

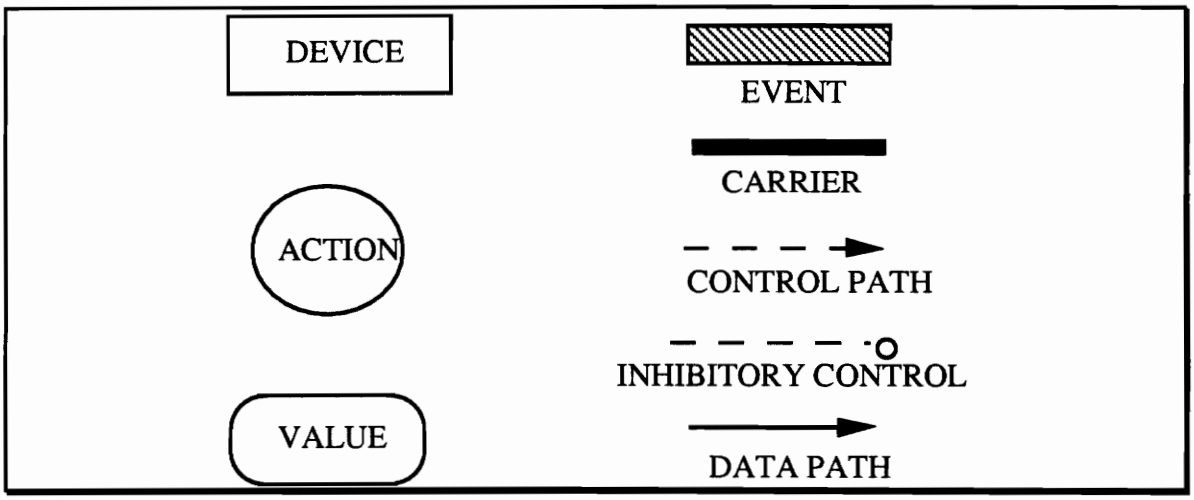


Figure 3.15. Icons used for Semantic Types

As seen in Figure 3.15, the concept type DEVICE is represented by a rectangle. The label in the rectangle is used to display the identifier used to refer to the DEVICE in the specification. If no name is specified, then the existence of the DEVICE is implied and the label used is the concept type. The exception in the representation of DEVICE is the

subtype CARRIER which is represented by solid line. The semantic types ACTION and STATE are represented by a circle. This notation is similar to denoting places in Petri net graphs. Labeling the circles with the associated verb or nominalized verb helps in symbolizing the action. The semantic type EVENT is represented using a Petri net transition (a heavy bar). The heavy bar representing EVENTS is called a *trans_bar* by the system. Table 1 provided the basic concept types used by the system.

The visual representation of relationship types varies widely. For example, behavioral relationship is represented by a heavy bar similar to the representation for EVENTS, whereas a containment relationship results in icons being drawn inside other icons. The heavy bar representing causal relationships is termed a *cause_bar* . The dotted arrow represents *control paths* (causal / temporal dependencies) and a solid arrow (called sol_arrow in the system) represents *data paths*. The bubbled dotted arrow represents an inhibtory control path and is a modification of the control path representation.

Table 2 provides the various conceptual relations used in the semantic model of the system. Relations *attr, det, freq, mod, name, neg, purp* and *quant* indicate the attributes of the concept and are appended to the referent label of that concept. Relations *agnt, obj, dest* and *inst* are represented by the icons corresponding to the related concept types. For example, an object can be a DEVICE (rectangle), ACTION (circle), or VALUE (rounded rectangle). Relation *cond* between two concepts is represented by a heavy bar between the two corresponding concept icons. The control dependency is indicated by drawing a dotted arrow from icons to the heavy bar and from heavy bar to other icons. Relation *loc* indicates the location of the object which is generally in a DEVICE concept, and this is represented by drawing the object concept icon inside the location concept icon. *Opnd* is always a VALUE and is represented with a rounded rectangle.

3.4. Interpretation Library

The Model Generator has an Interpretation Library (IL) which consists of canonical graphical representations for the conceptual graphs for the concept types in the vocabulary of the system. Details of these picture representations and the format in which they are stored in the database of the Model Generator are now explained.

3.4.1. Canonical Pictures

As mentioned earlier, every concept has one or more canonical conceptual graphs which represents the semantic attributes of the word. Using the representation scheme outlined above, we can pictorially represent the canonical graphs for concepts. These representations are called *canonical pictures* .

For example, for a canonical graph for INCREMENT given in Figure 3.6, the canonical picture is shown below. The *agent* of the root concept INCREMENT is an EVENT and its object is a VALUE. Since INCREMENT is a subtype of ACTION, it is represented with a circle. An EVENT is denoted by a heavy bar and a VALUE concept by a rounded rectangle. The control dependecies are indicated by the dotted arrows.

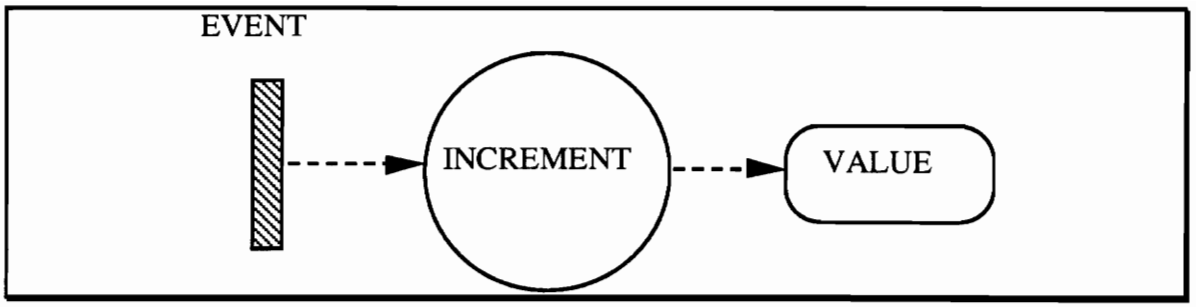


Figure 3.16. Canonical Picture for INCREMENT.

For the alternative canonical graph for INCREMENT (Figure 3.9), the canonical graph is shown in Figure 3.17. In this, the agent of the ACTION is a DEVICE concept. Since the ACTION is performed by a DEVICE concept, the circle representing INCREMENT is contained inside the icon representing a DEVICE concept (a rectangle).

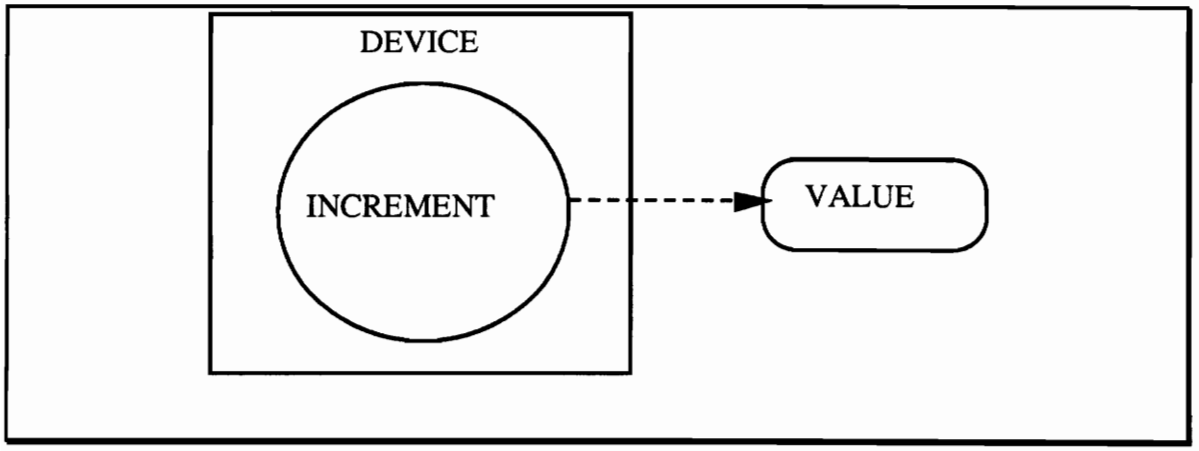


Figure 3.17. Alternative Canonical Picture for INCREMENT.

3.4.2. Scripts

The canonical pictures are stored in the Interpretation Library as a *script* of drawing commands for each of the canonical pictures. Each script line specifies either a drawing command for an icon or a drawing command for a connective between the icons.

The script line for an icon specifies,

(1) shape of icon,
(2) label field,
(3) the line index number,
(4) placement and dimension fields for the icon, and
(5) containments list for the icon.

The script line for connectives specifies,

(1) type of connective,
(2) label field,
(3) the line index number,
(4) line index of the source icon, and
(5) line index of the destination icon.

The *label* field of a script line is replaced by the referent and attributive strings of the corresponding concept. This is explained in greater detail in the next chapter. The *containments list* for an icon specifies the icons to be drawn inside it. Script for the

canonical picture for INCREMENT shown in Figure 3.16 is given in Figure 3.18. The fields labeled *length* and *height* are the dimension slots and are filled after determining the final size of the icon. The fields labeled x and y are the default placement coordinates which are used by the placement algorithm to calculate the final display coordinates.

```
[1] circle ( INCREMENT , x , y , length , height , { } )
[2] roundrect ( obj , x , y , length , height , { } )
[3] trans_bar ( agnt , x , y , length , height , { } )
[4] arrow ( name , from 3, to 1 , 0 , 0 , { } )
[5] arrow ( name , from 1, to 2 , 0 , 0 , { } )
```

Figure 3.18. Script for Canonical Picture for INCREMENT.

The curly brackets are used to identify the list of icons that need to be drawn inside the icon. This is illustrated by the script for the alternative canonical picture for INCREMENT (Figure 3.17), shown below, which indicates that the icon with line index 1 is drawn inside the icon with line index 2.

```
[1] circle ( INCREMENT , x , y , length , height , { } )
[2] rect ( agnt , x , y , length , height , { 1 } )
[3] roundrect ( obj , x , y , length , height , { } )
[4] arrow ( name , from 1, to 3 , 0 , 0 , { } )
```

Figure 3.19. Script for alternative Canonical Picture for INCREMENT.

The *arrow* command defines a connective between two icons. Since the default placement coordinates are not needed to draw connectives, these fields are used specify the source and

destination icon line index numbers. Also, dimensions for a connective are not needed and so the dimension slots contain zeros.

With each script, the corresponding canonical graph is attached. The format in which the canonical graph is stored in the Interpretation Library is shown in Figure 3.20.

```
$ concept $
(relation){basic concept type}[numbers]
.       .       .       .       .       .
(relation){basic concept type}[numbers]
```

Figure 3.20. Format for Canonical Graphs in the Interpretation Library.

The numbers in the square brackets are the line index numbers of the script lines which draw the representation for the relation before them. For example, the canonical graph for INCREMENT in Figure 3.9 is stored in the form shown below.

```
$ increment $
(agnt){device}[2]
(obj){value}[3 4]
```

Figure 3.21. Example Canonical in the Interpretation Library.

The graphical representation due to the relation *agnt* is drawn using the script line with index number 2 in the script shown in Figure 3.19. More discussion on this is presented in Section 4.3.

Chapter 4. The Mapping Engine

4.1. Overview

The Mapping Engine maps the input conceptual graph to a set of drawing commands called a script, that generates the graphical representation. The block diagram of the Mapping Engine is reproduced below. This module has been implemented in the C programming language.

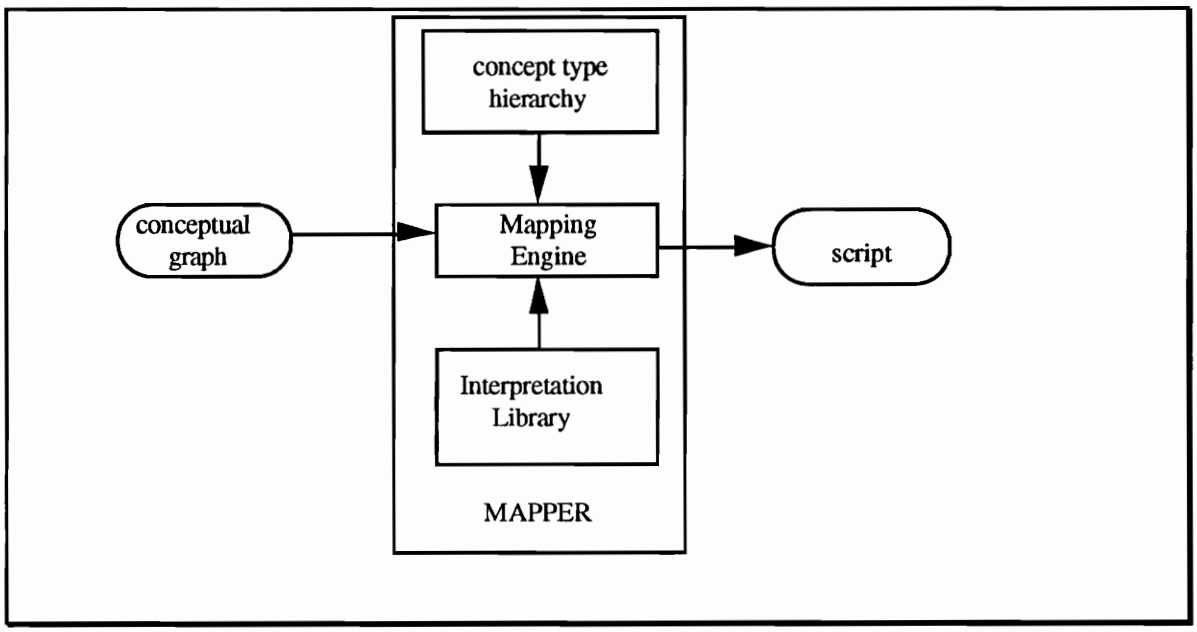


Figure 4.1. Block Diagram of the Mapping Engine.

Figure 4.2 shows the flow chart for the mapping process.

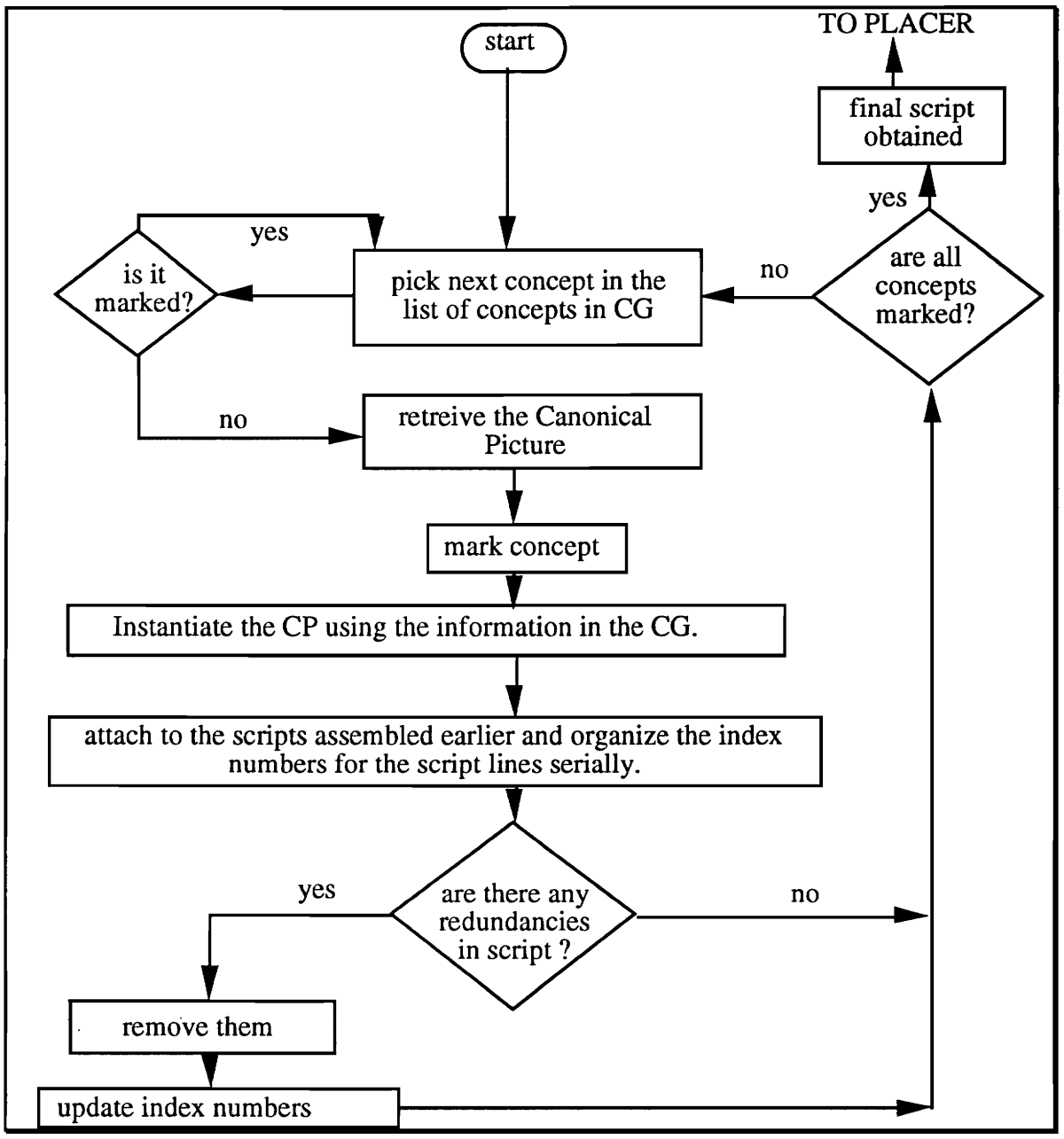


Figure 4.2. Flow Chart of the Mapping Process.

After the conceptual graph has been read into the internal data structures, canonical pictures for the concepts are retrieved from the Interpretation Library one by one. As each canonical picture is retrieved all its concept nodes, which are generics, are instantiated using the information provided in the conceptual graph. This involves replacing the label field in the script line with the concept type and its attributive strings. The attributive strings for a concept are found by conducting a depth-first search in the conceptual graph. After a canonical picture for a concept is retrieved, a flag is set for the concept. *Marking* the concept like this ensures that the canonical picture script for a concept is retrieved once only. The script is then appended to the previously assembled script. Before a new canonical picture is retrieved, any redundancies and multiple references in the assembled script are removed. After canonical pictures for all the concepts have been assembled the complete script for the graphical representation of the input conceptual graph is written to a file which is used by the Placer.

4.2. Internal Data Structures for Conceptual Graphs

This section provides the details of the data structures employed by the Mapping Engine. An illustrative example is also presented.

The concept node information from the conceptual graph is stored in a data structure called *contype*. The complete structure is presented below. The *cid* field stores the concept identification number (cid). The canonical conceptual graph index number is stored in the *gindx* field. A concept can be marked by setting the *mark* field as TRUE. The concept name and referent are found in *ctype* and *referent* fields respectively.

```
typedef struct concept{
        int cid;
        int gindx;
        int mark;
        char *ctype;
        char *referent;
        int supertype;
        struct relation *rel[15];
        Hints *Identifier_list;
} contype;
```

Each concept can have a maximum of 16 relations in this data structure. The maximum number of relations for a concept in the Interpretation Library is 4. Array *rel* stores the pointers to structures containing the relation information. *Identifier_list* points to the head of the list of attributive relations types. Attributive relations define the attributes and charecterstics of the concept. These have no graphical representation and are used as labels in the icon for the concept. Examples of such relations are, *name, quant, attr,* etc. Each node in this linked-list is defined as follows:

```
typedef struct hint{
        int id;
        char *rtype;
        struct hint *next;
}Hints;
```

Here *id* contains the cid of the related concept. The *rtype* field contains the name of the identifier relation.

The relation node information is stored in a data structure called *reltype*. It is defined as follows:

```
typedef struct relation{
        char *rtype;
        int conref;
        struct concept *c_cept;
}reltype;
```

The *rtype* field contains the type of the relation. The cid of the concept pointed to by the relation is stored in *conref*. The field *c_cept* points to the contype structure of the related concept.

After all the information has been read in, the complete graph resides in a data structure called *graphtype*. This contains the graph identification number (*gid*) and an array of concepts. Presently, the Mapping Engine can handle conceptual graphs with a maximum of 50 concepts. The largest conceptual graph which has been interpreted by the Model Generator consists of 24 concepts. The graph structure is type defined as follow:

```
typedef struct graph{
        int gid;
        int connum;
        contype *con[50];
}graphtype;
```

Let us look at how the internal organization of the program would be for the conceptual graph given below. The conceptual graph is for the sentence,

"the 8-bit data is loaded into the ACC register when STRB rises". (1)

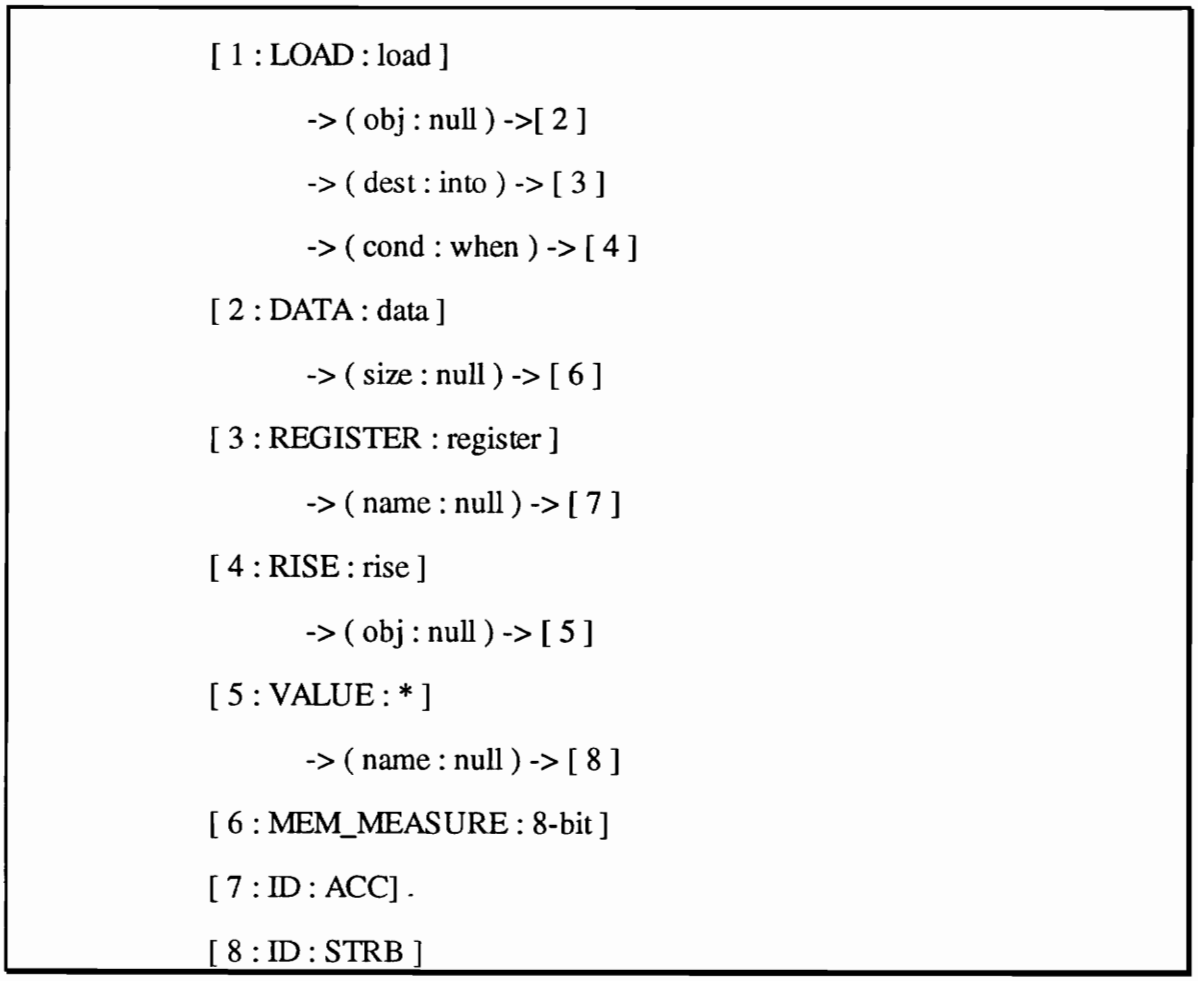


Figure 4.3. Conceptual graph for sentence #1

The data structure in the Mapping Engine for sentence #1 is shown in Figure 4.4 below.

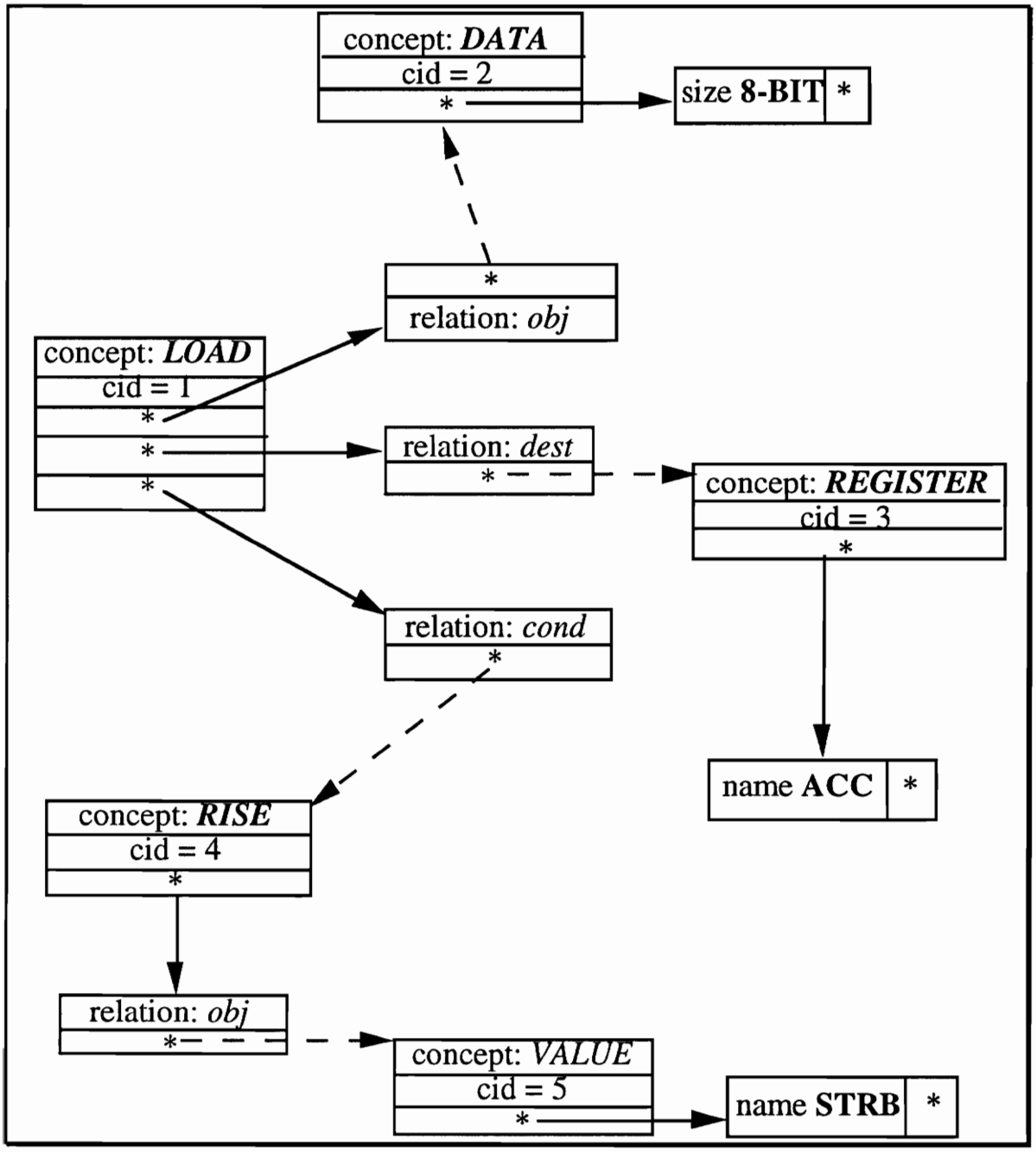


Figure 4.4. Internal Data Structure for sentence #1.

4.3. Retrieval of Canonical Pictures

Once the necessary information has been extracted, the Mapping Engine is ready to retrieve canonical pictures from the Interpretation Library.

Canonical Picture for each concept in the list of concepts is retrieved. Since there can exist multiple canonical pictures for a given concept, it is necessary to identify the correct canonical picture. For each canonical picture entry in the Interpretation Library, there is a canonical graph attached to it. By comparing the attached canonical graph with the information available for the concept and its relations, the Mapping Engine determines the canonical picture to be retrieved. The graph index is also compared to cross-check the correctness of the selected canonical picture.

The process of identifying the canonical picture to be retrieved for a concept is illustrated in Figure 4.5. The attached canonical graph provides the following information,

(i) the type of relations,

(ii) the type of concept pointed to by each relation, and

(iii) the line index numbers of the script lines that provide the concept representation.

For each relation in the canonical graph, the input conceptual graph is checked. This is necessary because only those concepts in the canonical graphs that are represented in the input sentence, are passed by the Semantic Analyzer in its output conceptual graph. If the related concept exists in the conceptual graph then the concept type is compared. If the latter comparison fails then the next canonical picture is picked up. If none of the canonical pictures are found to be correct then the system reports the error and exits.

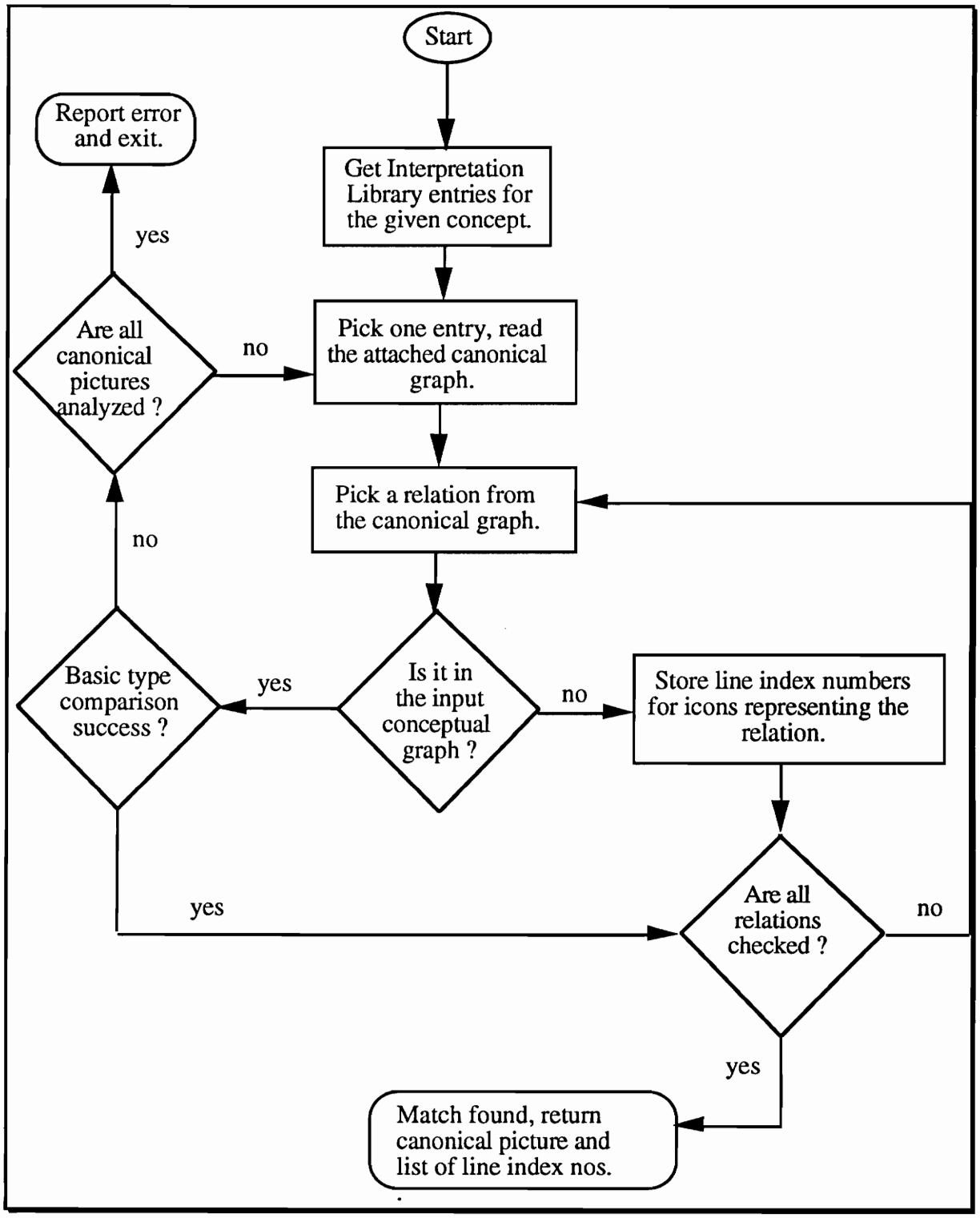


Figure 4.5. Flow chart for Canonical Picture selection.

If a particular concept in the canonical graph is absent in the input conceptual graph, then its corresponding graphical representation should not be displayed. Hence for each missing relation, the script lines for the relation representation are deleted from the canonical picture. To aid this, each relation in the canonical graph attached to the CP contains a list of line index numbers. These numbers identify the script lines that provide the draw commands for the graphical representation for the relation. On finding an absent relation, the corresponding list is used to delete the script lines from the canonical picture.

For example, the Interpretation Library entry for the concept LOAD is shown below. The pictorial form of the attached canonical graph is on left and the canonical picture is on the right.

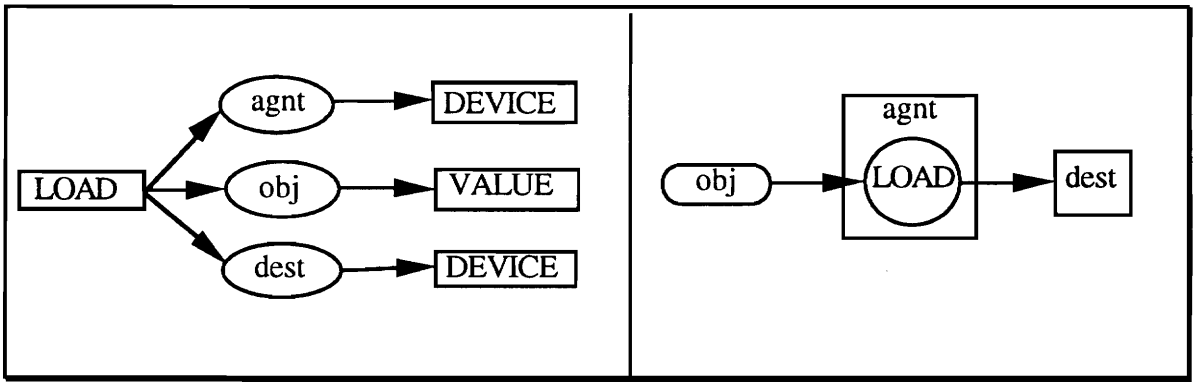


Figure 4.6(a). Canonical Graph and Canonical Picture for LOAD.

The script for the canonical picture is shown in Figure 4.6(b).

```
[1] circle ( LOAD, x , y , length , height , { } )
[2] rect ( agnt, x , y , length , height , { 1 } )
[3] roundrect ( obj, x , y , length , height , { } )
[4 ] rect ( dest, x , y , length , height , { } )
[5] solarrow ( name , from 3, to 1 , 0 , 0 , { } )
[6] solarrow ( name , from 1, to 4 , 0 , 0 , { } )
```

Figure 4.6(b). Script of Canonical Picture for LOAD.

The canonical conceptual graph attached to the canonical picture in the Interpretation Library is in the form shown in Figure 4.7,

```
$ load $
(agnt) {device} [2]
(obj) {value} [3 5]
(dest) {device} [4 6]
```

Figure 4.7. Canonical Graph for LOAD in the Interpretation Library.

The numbers between the square brackets are the line index numbers corresponding to the relation before them. Thus, the graphical representation for the relation *agnt* is the drawing command with line index number 2. If we assume that this relation is absent in the input conceptual graph, then the drawing command for *rectangle* is not retrieved by the Mapping Engine. So, for this case, the final script retrieved is shown in Figure 4.8(a) and its graphical form shown Figure 4.8(b).

[1] circle (LOAD, x , y , length , height , { })

[2] roundrect (obj, x , y , length , height , { })

[3] rect (dest, x , y , length , height , { })

[4] sol_arrow (name , from 2, to 1 , 0 , 0 , { })

[5] sol_arrow (name , from 1, to 3 , 0 , 0 , { })

Figure 4.8(a). Script for CP retrieved for LOAD in sentence #1.

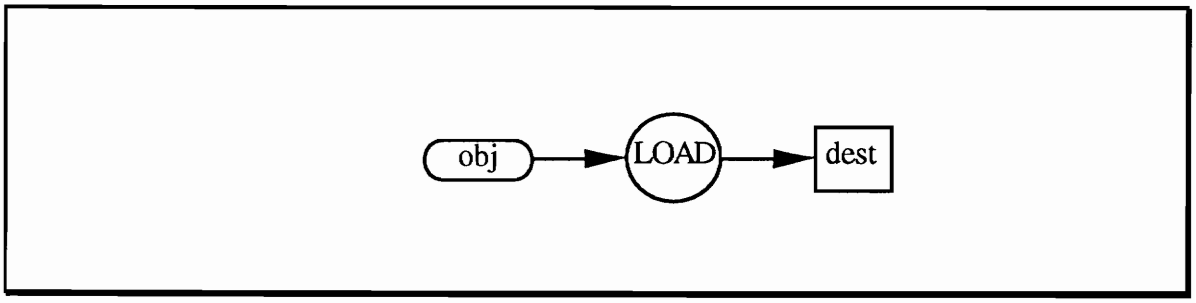


Figure 4.8(b). Canonical Picture retrieved for LOAD in sentence #1.

4.4. Attachment of Canonical Pictures

As each canonical picture is retrieved from the Interpretation Library, it is appended to the previously assembled picture. Before appending, each generic concept node in the canonical picture is instantiated using the concept information extracted from the input conceptual graph.

It will be recalled that canonical pictures have script lines for drawing icons representing the concepts related to the root concept in *generic* form. This means that the label field in the script line contains the *type* of the relation. Instantiation involves

replacing the basic relation type, in the label field, with the concept information to be displayed. This is outlined by the flow chart shown in Figure 4.9.

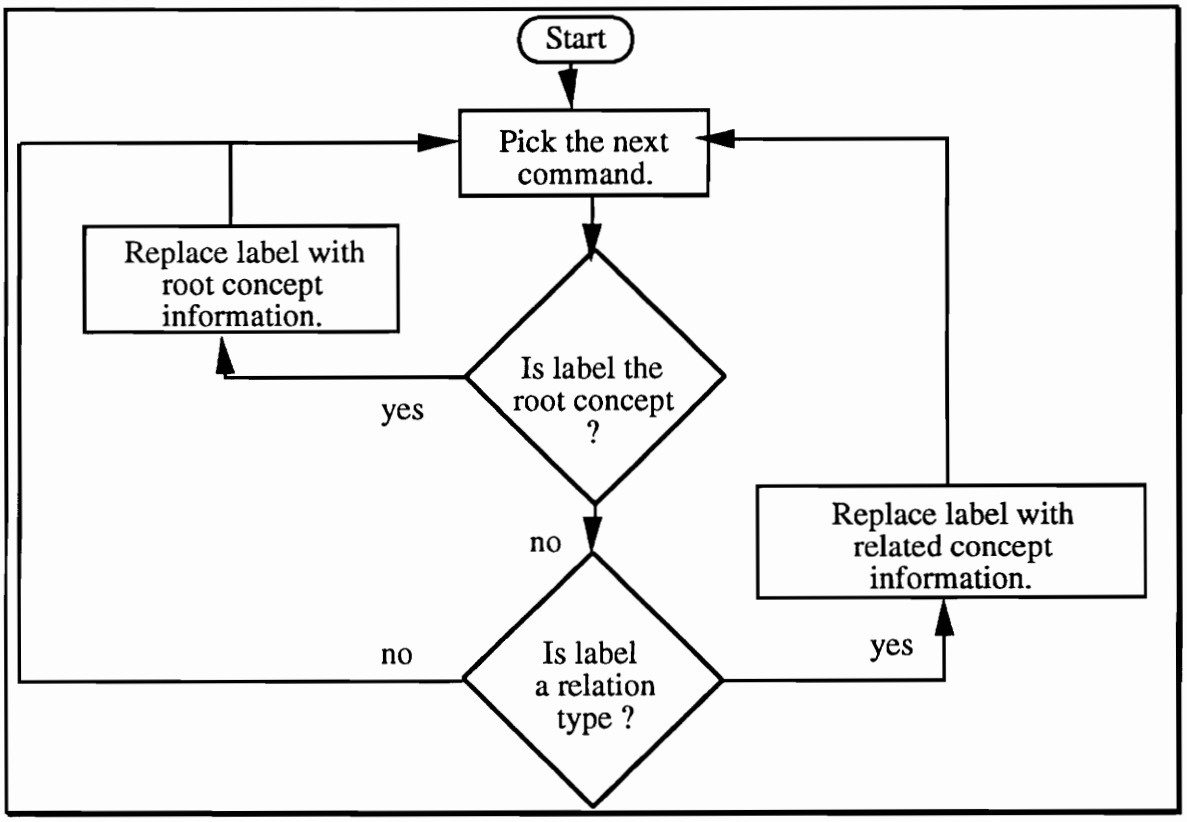


Figure 4.9. Flow Chart for Instantiation process of Canonical Pictures.

Concept information mentioned above includes -

concept type,

referent for the concept,

all attributive words for the concept and

cid.

The canonical picture for LOAD shown in Figure 4.6 can be instantiated by using the information supplied in the conceptual graph in Figure 4.3. The script for the

instantiated canonical picture is shown below. The instantiated labels contain the concept types in lower case and this is the form displayed in the final graphical representation.

[1] circle (**load: 1**, x , y , length , height , { })

[2] roundrect (**data: 8-bit: 2**, x , y , length , height , { })

[3] rect (**register: ACC: 3**, x , y , length , height , { })

[4] sol_arrow (name , from 2, to 1 , 0 , 0 , { })

[5] sol_arrow (name , from 1, to 3 , 0 , 0 , { })

Figure 4.10. Instantiated Canonical Picture for LOAD.

There are some relations like *cond* and conjunctions *and* and *or*, that are not present in the canonical graph database for concepts. These are attached at the time of semantic analysis of the parse trees. Since these are absent in canonical graphs, their graphical interpretations are not present in the Interpretation Library. The Mapping Engine must generate new drawing commands (or modify old ones) to represent such relations. After instantiating the generic concept nodes, handling for such relations is performed.

The *cond* relation specifies a condition on an ACTION or EVENT. The graphical representation for this relation is a heavy bar with arrows indicating control dpendency. For example, if action A is a condition for action B, the graphical representation would be as shown in Figure 4.11.

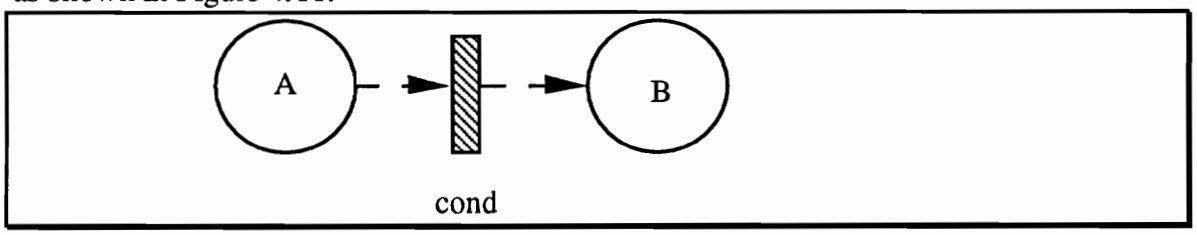


Figure 4.11. Example of *cond* relation representation.

The script after appending the script lines needed to represent the *cond* relation is shown in Figure 4.12(a) below.

[1] circle (**load: 1**, x , y , length , height , { })

[2] roundrect (**data: 8-bit: 2**, x , y , length , height , { })

[3] rect (**register: ACC: 3**, x , y , length , height , { })

[4] sol_arrow (name , from 2, to 1 , 0 , 0 , { })

[5] sol_arrow (name , from 1, to 3 , 0 , 0 , { })

[6] cause_bar (**cond**, x , y , length , height , { })

[7] arrow (name , from 6, to 1 , 0 , 0 , { })

[8] arrow (name , from 9, to 6 , 0 , 0 , { })

[9] circle (**rise: 4**, x , y , length , height , { })

Figure 4.12(a). Canonical Picture for LOAD with *cond* relation.

Definite articles like *the* and indefinite articles like *a* and *an*, grouped under the conceptual relation *determiner*, have no graphical representation and they are reflected in the concept referent label.

An exception in the graphical representation for concepts is for IS, a subtype of STATE concept. This concept can be used to equate two DEVICES. Sentence "*the processor is a 16-bit device*" indicates that the PROCESSOR (a DEVICE) is also a specific device (16-bit). The representation for this sentence should have only one rectangle since PROCESSOR and 16-bit DEVICE are two different names for the same DEVICE concept. This is achieved by including both names in the label of a rectangle. Alternatively,

"*STRB is high*" indicates a logical equivalence between the value of the STRB signal and a high value. This is represented by connecting the icons for STRB and a high value with a circle containing an "=" sign.

Once the instantiation is done, the canonical picture script is appended to the previously assembled script.

```
[1] circle ( load: 1, x , y , length , height , { } )
[2] roundrect ( data: 8-bit: 2, x , y , length , height , { } )
[3] rect ( register: ACC: 3, x , y , length , height , { } )
[4] solarrow ( name , from 2, to 1 , 0 , 0 , { } )
[5] solarrow ( name , from 1, to 3 , 0 , 0 , { } )
[6] cause_bar ( cond, x , y , length , height , { } )
[7] arrow ( name , from 6, to 1 , 0 , 0 , { } )
[8] arrow ( name , from 9, to 6 , 0 , 0 , { } )
[9] trans_bar ( rise: 4, x , y , length , height , { } )
[10] circle ( rise: 4, x , y , length , height , { } )
[11] roundrect ( value: STRB: 5, x , y , length , height , { } )
[12] arrow ( name , from 11, to 10 , 0 , 0 , { } )
```

Figure 4.12(b). Script for RISE appended to Script for LOAD.

Figure 4.12(b) above shows the state when the canonical picture for concept RISE is appended to the previously assembled script, namely, the canonical picture for concept LOAD. The index numbers of the script lines are also reorganized serially.

4.5. Removal of Multiple References

After appending the new canonical picture, it is necessary to remove multiple references in the assembled script.

These redundancies can be in the form of the following:

(i) multiple draw commands for icons for the same representation, and

(ii) multiple draw commands for connections between a pair of icons.

In Figure 4.12 it can be seen that the assembled script contains redundancies. The draw command for the icon representing RISE occurs twice.

The process of removing multiple references is shown in Figure 4.13. Each script command is checked for multiple references. If any other line in the script specifies the same draw command and has the same label then it is a multiple reference and is removed from the script. Before the script line is removed, it is ensured that all the connectives of the *removed* script command are added to the connectives list of the *removing* script command. The containments list of the remaining command is modified to include the containments of the removed commands.

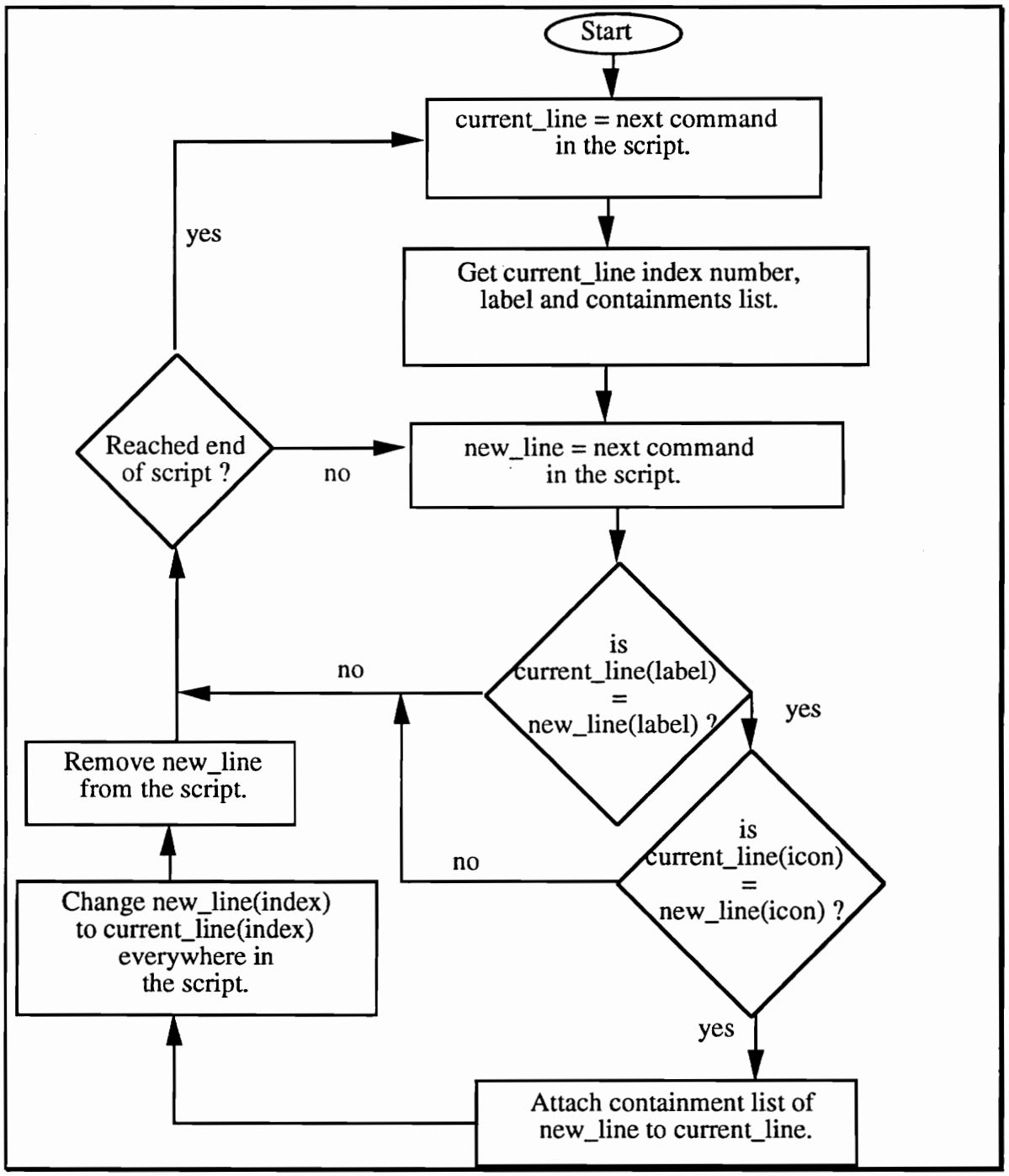


Figure 4.13. Flow chart for the process of Removing Multiple References.

By continuing this process for each line in the script, all the redundancies in the script can be removed. On completion of this process, each line, for both icons and connectives, is unique in the script.

After the multiple reference is removed from the script in Figure 4.12, we get,

```
[1] circle ( load: 1, x , y , length , height , { } )
[2] roundrect ( data: 8-bit: 2, x , y , length , height , { } )
[3] rect ( register: ACC: 3, x , y , length , height , { } )
[4] solarrow ( name , from 2, to 1 , 0 , 0 , { } )
[5] solarrow ( name , from 1, to 3 , 0 , 0 , { } )
[6] cause_bar ( cond, x , y , length , height , { } )
[7] arrow ( name , from 6, to 1 , 0 , 0 , { } )
[8] arrow ( name , from 9, to 6 , 0 , 0 , { } )
[9] circle ( rise: 4, x , y , length , height , { } )
[10] roundrect ( value: STRB: 5, x , y , length , height , { } )
[11] arrow ( name , from 10, to 9 , 0 , 0 , { } )
```

Figure 4.14. Script after removal of Multiple Reference.

It should be noted that the connective to the deleted script line has been re-directed. If the deleted script line had a list of containments, this list would have been added to the remaining script line for RISE. Figure 4.15 shows the graphical representation for sentence #1.

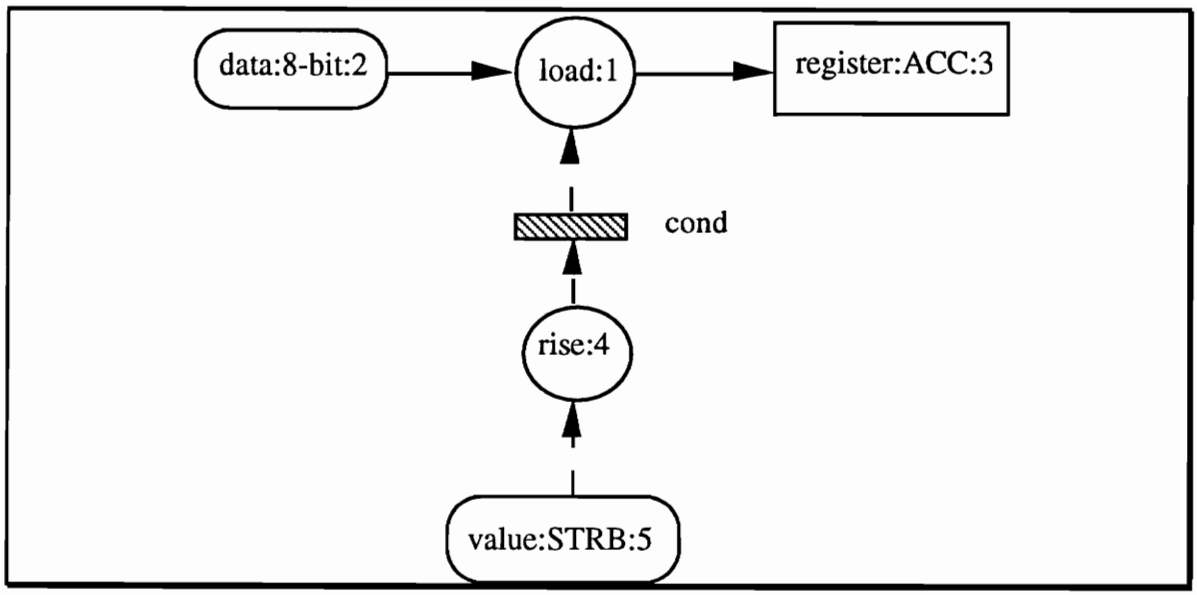


Figure 4.15. Graphical Representation for sentence #1.

Chapter 5. The Placer

5.1. Overview

After the script for the graphical representation has been assembled, the Model Generator invokes the Placer for generating screen coordinates for the various icons and displaying them. The block diagram of Placer is repreoduced below.

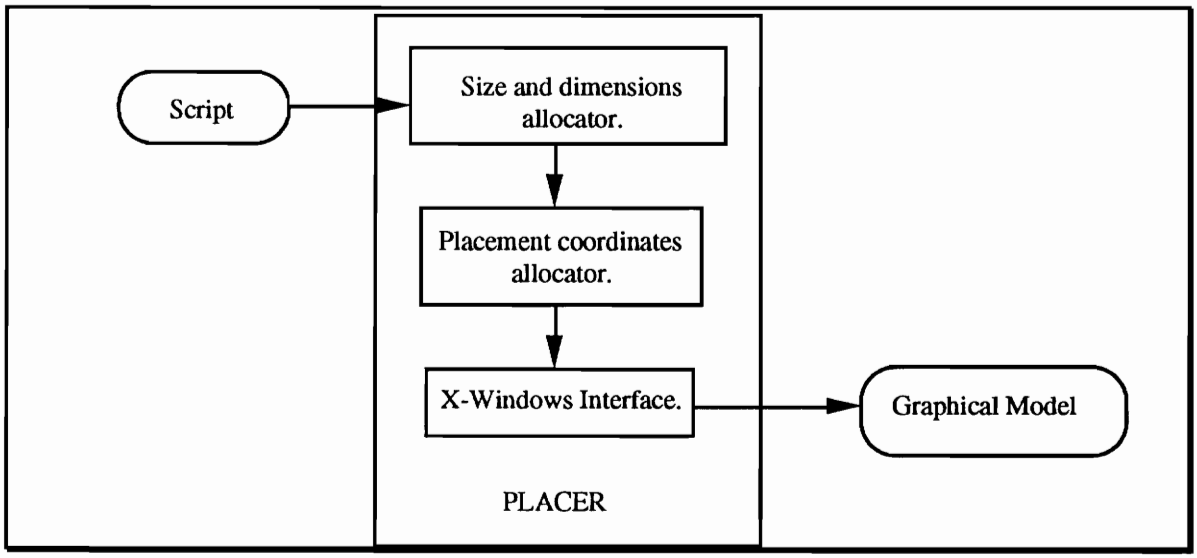


Figure 5.1. Block Diagram of the Placer.

Figure 5.2 shows the flow chart for the process of placement.

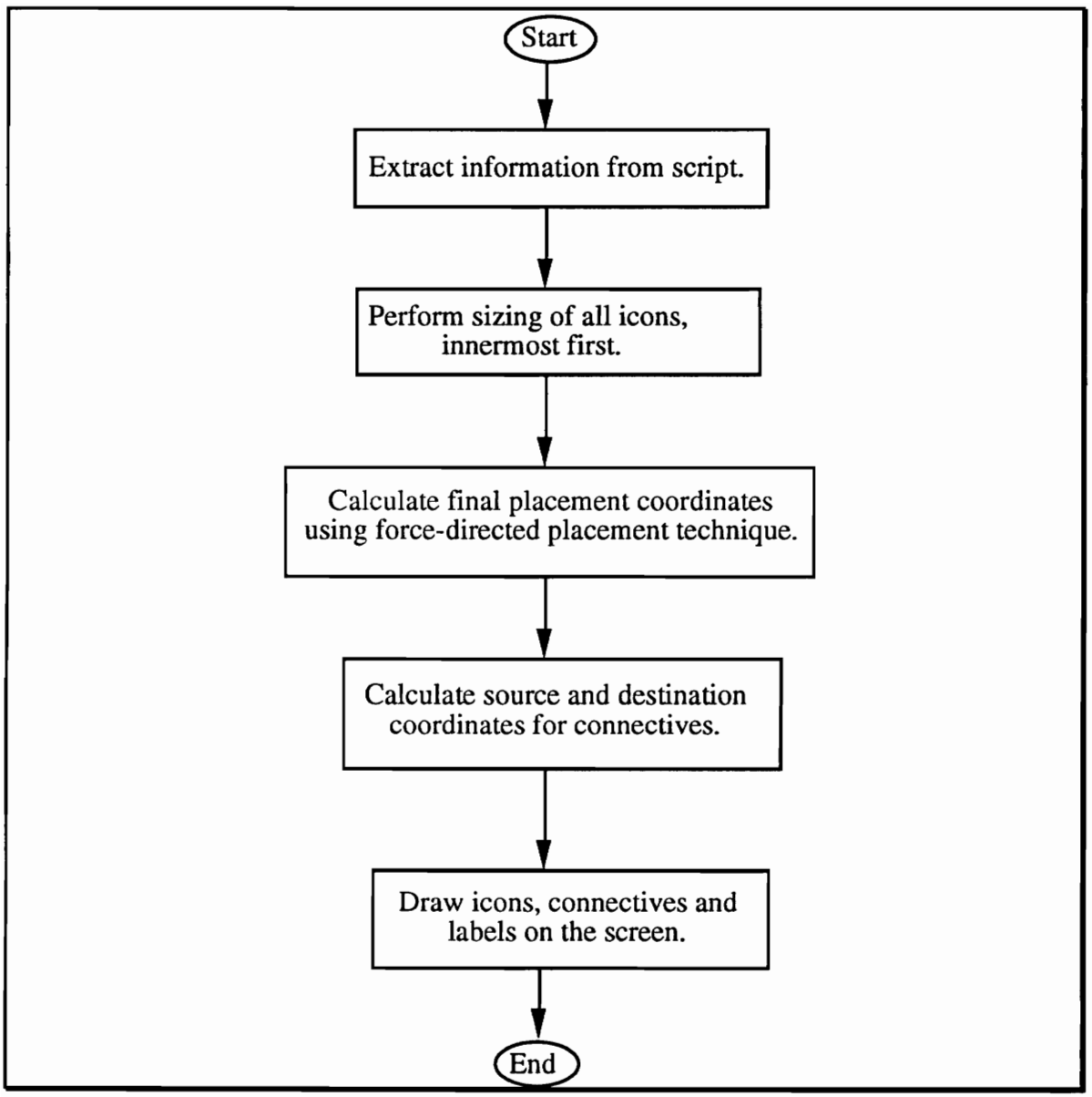


Figure 5.2. Flow Chart of the Placement Process.

The script lines are read one by one and the information is stored in the internal data structures. After this, sizing of each icon is performed using its list of containments and the size of its label string. This is a recursive process begining with the sizing of the innermost icon first. Once the sizing is finished, the default base coordinates of the

outermost icons are supplied to a force-directed placement routine. This routine returns the final placement coordinates for all the outermost icons. The inner icons are then placed. After the placement coordinates of all the icons have been calculated, the connectives are drawn.

The Placer uses its X-Window interface to draw on the screen. The Placer module has been written in the C programming language under the X-Windows environment.

5.2. Internal Data Structures

Provided in this section are the details of the data structures used by the Placer. The various data elements are presented and explained. The main data structure is explained in two parts, one pertaining to information extraction from scripts, and the other pertaining to the force-directed placement.

The basic data structure used by the Placer is called *drawtype*. It is shown below. The field *line_num* specifies the line index number of the script line. The *type* of the icon or connective is stored in the type field. The *name* field points to the label of the icon. Fields *disp_x*, *disp_y* and *distance* are used by the placement algorithm. Field *in_num* contains the number of icons in the containment list. The containment list is in the form of a linked-list. The linked-list storing the containment information is defined by the elements *head* and *tail* of the data structure. *Head* points to the start of the containment list and *tail* points to the end of the list. The *tail* of the linked-list is useful in adding new nodes in the list.

```
typedef struct draw{
        int line_num;
        int type;
        char *name;
        Dimension width, length;
        Position x, y;
        Position disp_x, disp_y;
        int distance;
        int in_num;
        dnode *head;
        dnode *tail;
}drawtype;
```

Each node in the linked-list is a structure called *dnode*. This structure is defined as follows,

```
typedef struct inside_node{
        struct draw *my_ptr;
        struct inside_node *next;
}dnode;
```

All the icons and connectives are stored in a global array of *drawtype* structures. The Placer presently can handle upto 100 icons and connectives.

For the script given below, the internal organization of the memory elements of the Placer is shown in Figure 5.4. The default base coordinates (x_i, y_i) and dimensions (length, height) have not been shown here.

[1] rect (processor: 80486, x_1, y_1, $length_1$, $height_1$, { 2, 3, 4 })

[2] rect (cache_1: on-chip: 8K, x_2, y_2, $length_2$, $height_2$, { })

[3] rect (processor: main, x_3, y_3, $length_3$, $height_3$, { })

[4] rect (co-processor, x_4, y_4, $length_4$, $height_4$, { })

Figure 5.3. An Example Script.

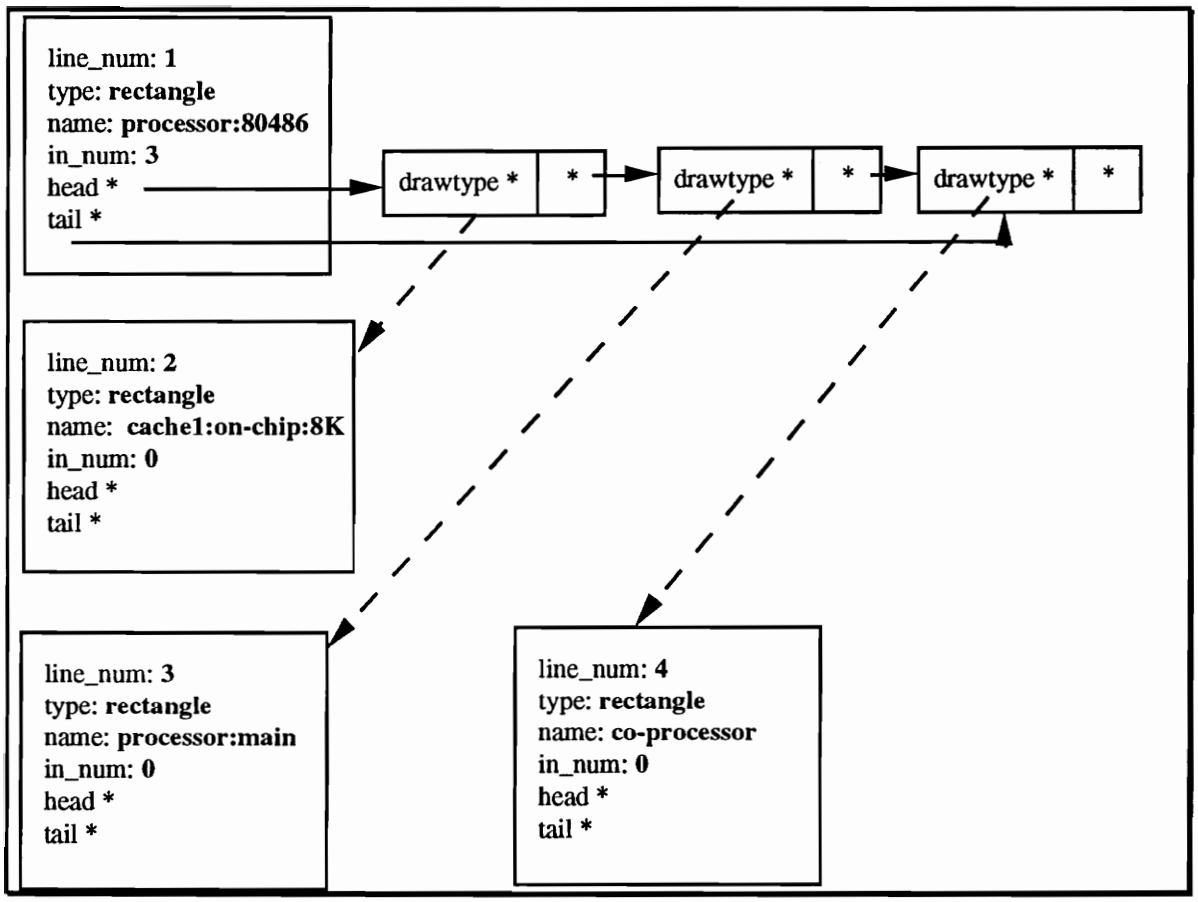


Figure 5.4. Internal Data Structure for the Script in Figure 5.3.

5.3. Icon Sizing

Once the internal data structures have been filled up, the Placer calculates the dimensions for each icon.

The main sizing routine is recursive. Since icons can reside inside other icons, it is necessary to obtain the dimensions of the innermost icons first. The most level of recursion encountered so far by the Model Generator is four. To calculate the dimensions

of an icon that does not contain any other icon, the dimensions of the bounding box of the label string is used. This is accomplished by the routine *find_string_size*() in the Placer. Each icon size is the size of its bounding rectangle. For the heavy bars used to represent EVENTS and causal dependencies, default coordinates are used and their labels drawn on the side. For the rest, labels are drawn at a fixed offset from the base coordinates.

5.4. Force Directed Placement

After the dimensions have been calculated, the Placer uses a force-directed placement routine to generate screen coordinates for the icons.

5.4.1. The Physical Model

The force-directed placement routine models the graph as a physical system and uses a laws of equilibrium of forces to obtain the final layout. In this model, the vertices are replaced with rings and the edges with springs. This mechanical system is given an initial configuration (vertex coordinates) and released. The forces exerted by the springs cause the rings to be placed in an equilibrium configuration having minimal energy state. The spring constant ensures that the vertices are not too far apart or close together.

The law governing the forces exerted by springs is known as the Hooke's law. It is a macroscopic approximation of the behavior of springs and can be summarized as follows. If a spring is compressed or extended and released, it returns to its original or natural length, provided the displacement is not too great [21]. It can be seen that for a

small displacement x, the force exerted by the spring is proportional to x. This can be written as,

$$F_X = -k\,x, \qquad \text{(A)}$$

where k is called the spring constant.

Equation A uses linear springs. For complex graphs these springs work poorly because they are unable to recover from bad initial placement [7]. Intuitively, if two vertices are too far apart or too close together, the more 'voilent' should be the forces correcting their placement. The following functions were proposed by Fruchterman and Reingold, and are used by the Placer.

$$F_r = d^2 / k, \text{ and}$$
$$F_a = -k^2 / d, \qquad \text{(B)}$$

where **d** is the distance between the two vertices and **k** is the optimal distance between vertices.

For a display window of area A and containing N vertices, the empty area around each vertex is A/N. The optimal distance between two vertices is the radius of this empty area. The value of **k** can be calculated using the following formula [7].

$$k = C \ \text{sqrt}(\text{area of window} / \text{number of vertices}), \qquad \text{(C)}$$

where C is constant determined experimentally. Logarithmic springs, proposed by Eades [5], generate the same results as the springs in Equation B, but were found to lack in terms of computational efficiency [7].

5.4.2. The Placement Algorithm

The placement algorithm can be summarized by the flow chart shown in Figure 5.5. There are five steps to each iteration: calculate the effect of repulsive forces on each vertex due to every other vertex in the graph, calculate the effect of attractive forces between vertices connected by edges, limit the total displacement of each vertex by a predetermined maximum value, prevent the vertices from going outside the window and reduce the maximum permissible displacement for the next iteration.

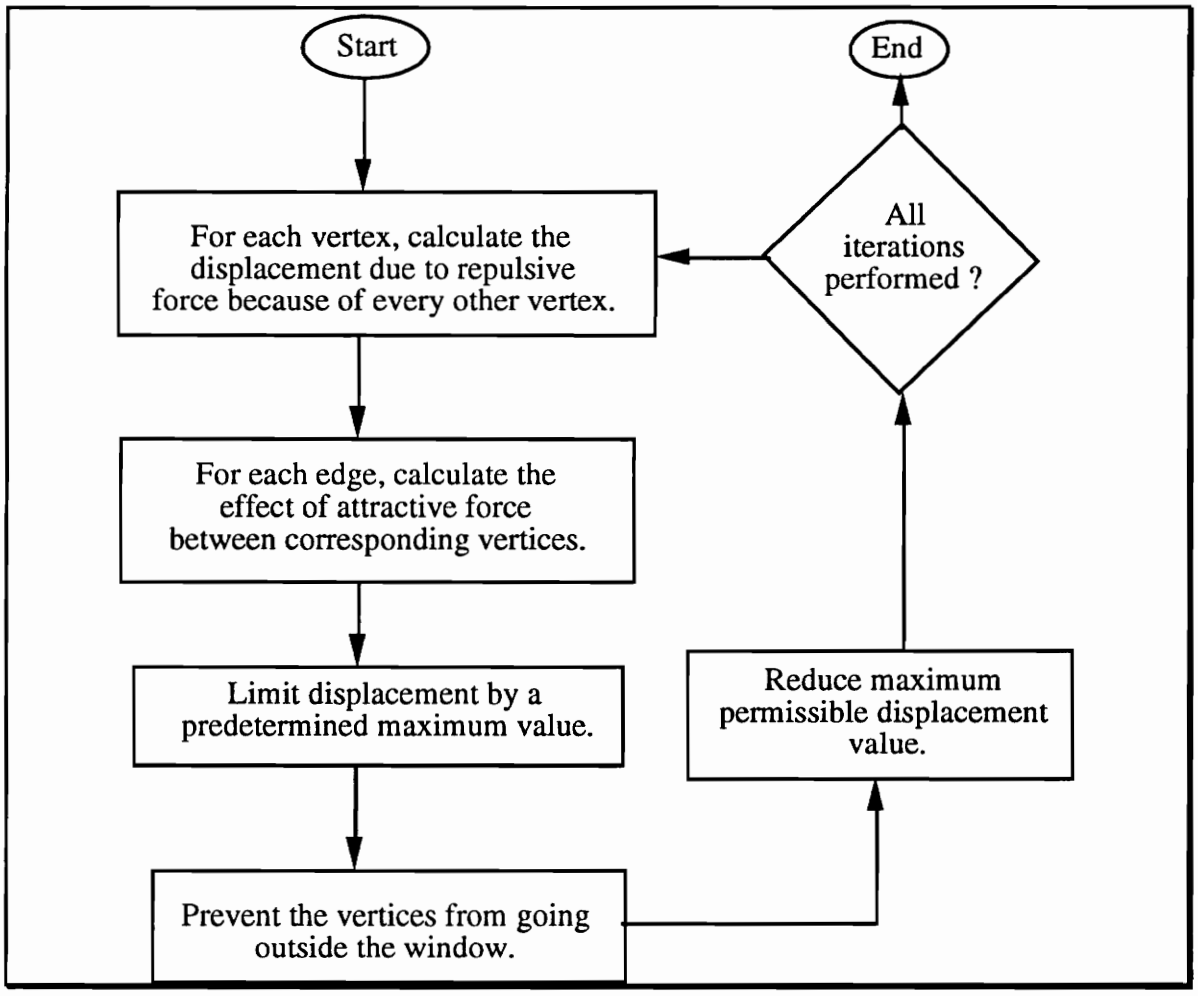


Figure 5.5. Flow Chart for the Force-Directed Placement Technique.

Let us now look at the details of the steps involved in each iteration. A vertex represents an icon that is not contained in any other icon. Edges represent all the connectives in the graph. Each vertex is assumed to have two vectors: position and displacement. The position vector of a vertex, v, is referred to with v.pos.x and v.pos.y. The displacement vector for v has v.disp.x and v.disp.y. Each edge is an ordered pair of vertices, say u and v, and so corresponding displacement vectors are e.u.disp and e.v.disp. Presented below is the detailed algorithm.

The number of iterations used by the Placer are 100. It has been found that 100 iterations are sufficient to obtain an equilibrium state for most graphs [5]. The value of the constant C was experimentally determined and C=1 was found suitable.

area = W * L; { W and L are the width and the length of the window }
G = (V, E); { graph G consists of vertex set V and edge set E }
{ Initial positions are assigned to the vertices }
T = Initial maximimum permissible displacement;
$F_a(d) = d^2 / k$;
$F_r(d) = -k^2 / d$;
k = sqrt (area / number of vertices);

for i = 1 **to** 100 **do**
    { Calculate repulsive forces }
    for v **in** V **do**
        v.disp.x = 0;

```
        v.disp.y = 0;
        for u in V do
            if ( u not equal v ) do
                D.x = v.pos.x - u.pos.x;
                D.y = v.pos.y - u.pos.y;
                v.disp.x = v.disp.x + (D.x / |D.x| ) * Fr ( |D.x|);
                v.disp.y = v.disp.y + (D.y / |D.y| ) * Fr ( |D.y|);
            end if;
        end for;

    { Calculate attractive forces }
    for e in E do
        { each edge is an ordered pair of vertices .v and .u }
        D.x = e.v.pos.x - e.u.pos.x;
        D.y = e.v.pos.y - e.u.pos.y;
        e.v.disp.x = e.v.disp.x - (D.x / |D.x|) * Fa (|D.x|);
        e.v.disp.y = e.v.disp.y - (D.y / |D.y|) * Fa (|D.y|);
        e.u.disp.x = e.u.disp.x + (D.x / |D.x|) * Fa (|D.x|);
        e.u.disp.y = e.u.disp.y + (D.y / |D.y|) * Fa (|D.y|);
    end for;

    { Limit displacement by T }
    { and prevent the vertices from going out of the frame }
    for v in V do
        v.pos.x = v.pos.x + (v.disp.x / |v.disp.x| ) * min (v.disp.x, t);
        v.pos.y = v.pos.y + (v.disp.y / |v.disp.y| ) * min (v.disp.y, t);
```

```
            v.pos.x = min (W/2, max(-W/2, v.pos.x);
            v.pos.y = min (L/2, max(-L/2, v.pos.y);
        end for;
        { Cool temperature t so that maximum possible displacement decreases }
        T = reduce(T);
end for;
```

The displacement vector is initialized to zero for each vertex during the calculation of repulsive forces. The position vector uses the default coordinates supplied in the script. D (D.x and D.y) is called the *difference* vector. |D| is the absolute value of the vector. Function f_a and f_r have already been defined earlier.

First, the repulsive forces are calculated between all the vertices. Then the effect of these forces is translated to corresponding displacement. Calculating repulsive forces on all vertices has the effect of scattering all the vertices. Then the attractive forces between adjacent vertices are calculated and their effect translated to displacement. This causes only the connected vertices to come closer.

The cumulative displacement of each vertex is limited to a predetermined maximum value, T. This ensures that vertices that are placed too close together or too far apart do not face large coorective forces for too many iterations. After each iteration this maximum value is decreased using a function that reduces the maximum permissible displacement (*reduce*(T)). The idea behind this is that as the layout becomes better the adjustment is made finer. The initial value chosen for T is 500 for the Placer. The value is made to

decay to 0 in steps of 5 per iteration. The choice of the initial maximum value was determined by experimentation.

Finally, vertices are prevented from being displaced out of the frame. The frame, the window in our case, is 500 by 500 units in size. The above algorithm assumes the center of the frame as the origin of the cartesian plane. Since X-Window routines use the top left corner as origin, all the coordinates need to be normalized before being used for display.

After the base coordinates for the outermost icons have been determined the inner icons are placed. Since the number of icons inside other icons is typically small, a simplistic placement is performed. The inner icons are placed side by side. Using force-directed placement for inner icons was avoided because modeling outer icons as frames becomes difficult and the time complexity increases.

5.5. Attach Points

After the base coordinates have been determined, all the necessary information needed to draw the icons is available. Now, calculation for the source and destination coordinates for the connectives is performed.

Since we are not using a grid-based system, the connectives follow straight lines. This means that the connectives are drawn straight from the source icon to the destination icon. The points where the straight line between the centers of the two connected icons

intersects the boundary of the icon are determined. These *attach points* are stored in the drawtype structure for the connective between the icons.

5.6. X-Windows Interface

After the final placement coordinates have been determined the X-Window Library (Xlib) functions are invoked. Some of the icons used by the Model Generator can directly be drawn using the existing Xlib functions [24, 25]. The remaining functions have been developed using the existing functions.

The X-Window interface of the Placer also allows the user to drag icons using the mouse to increase the aesthetics of the generated picture. This is necessary because some connectives can cross each other and other icons. When an icon is being dragged, all the inner icons are also dragged. Also, all the connectives of the dragged icons remain attached. This necessitates recalculation of the 'attach' points.

Figure 5.6 shows the graphical model for the sentence ,

" initializing the memory causes all the data in it to be deleted ". (2)

The figure shows the picture before the application of the placement algorithm. Figure 5.7 shows the same graphical model after the placement routine has been applied. The numbers in the labels after the last ":" are the cid numbers of the corresponding concepts in the conceptual graph.

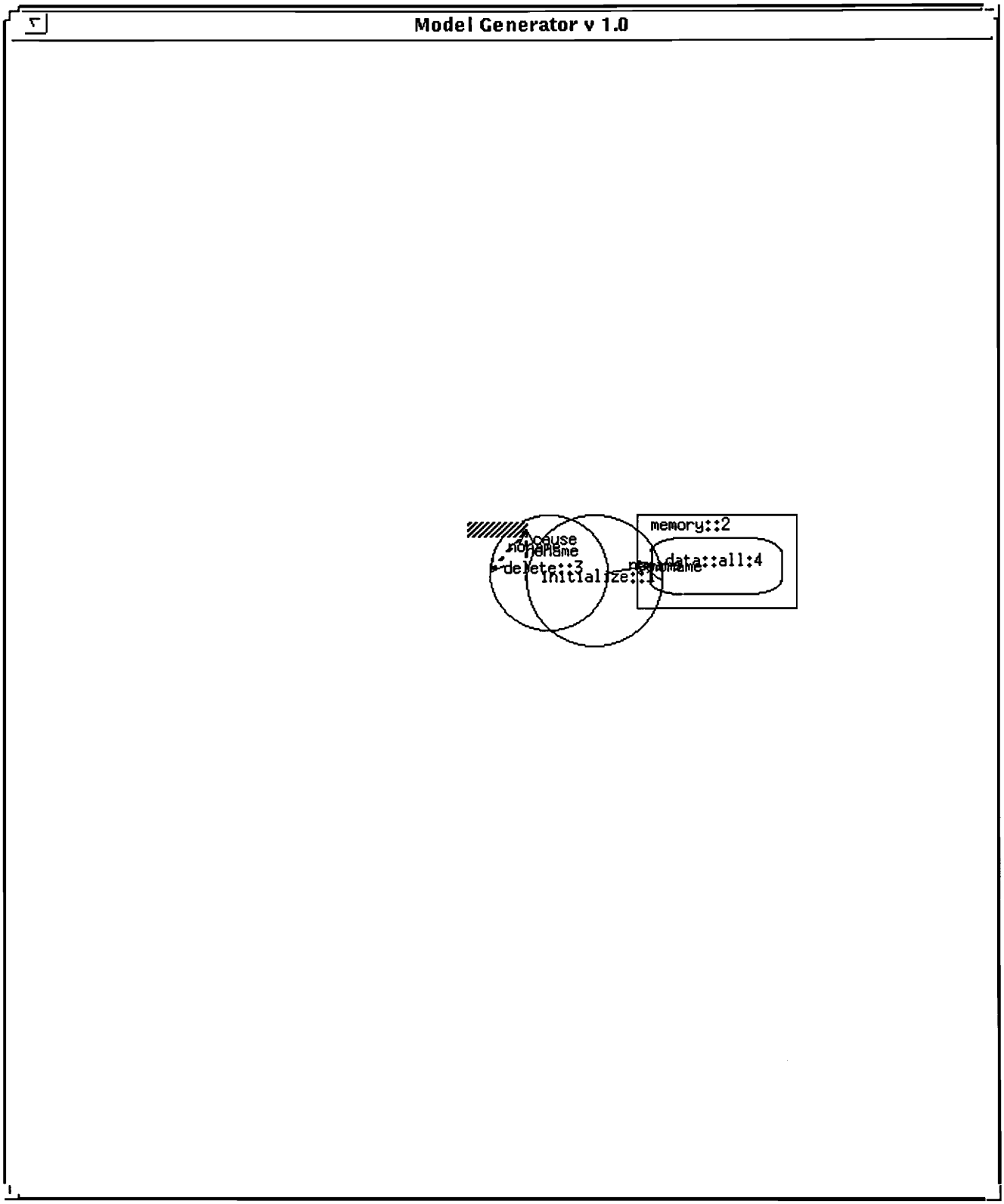


Figure 5.6. Graphical Model for Sentence #2 before Placement.

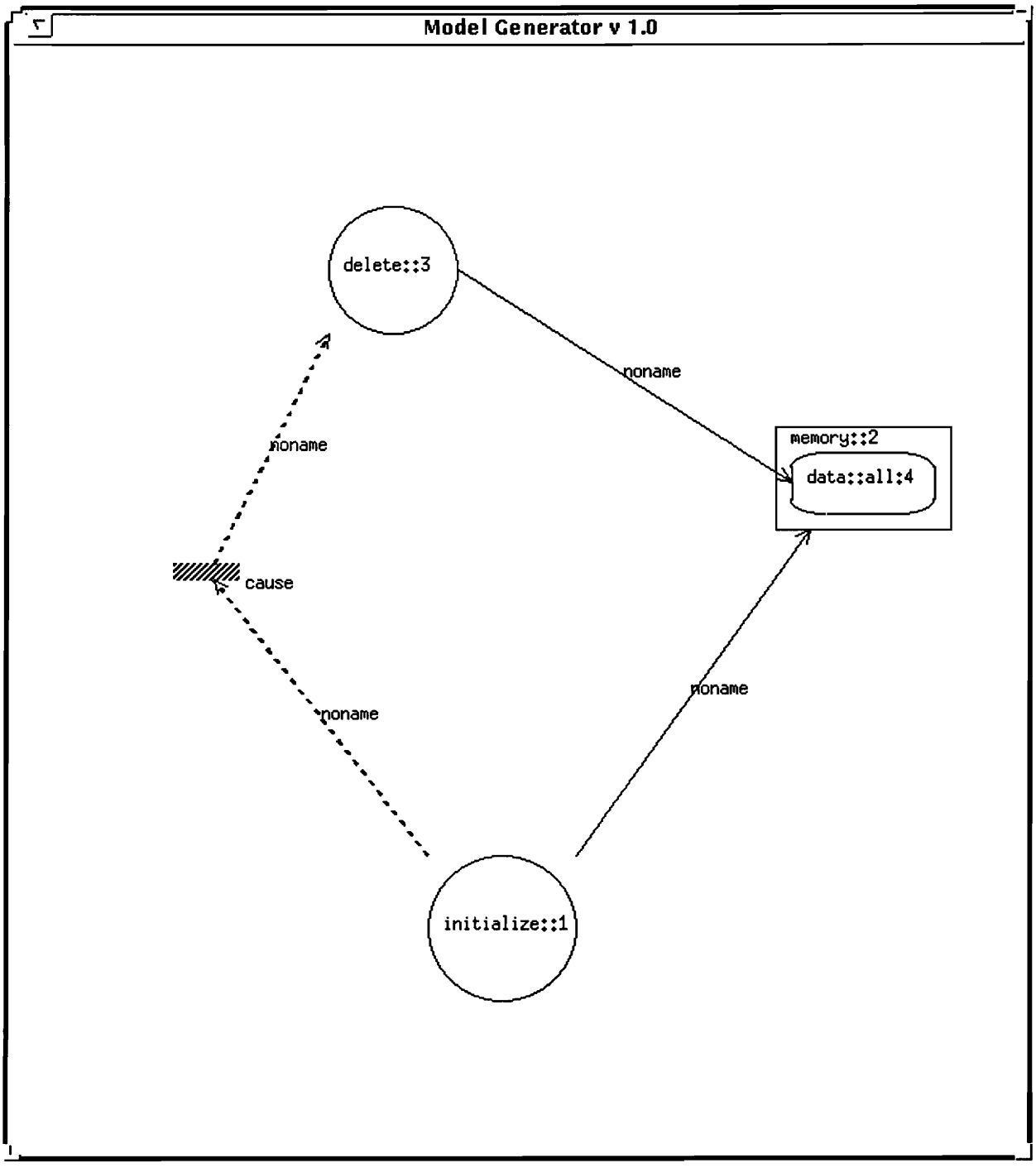


Figure 5.7. Graphical Model for Sentence #2 after Placement.

Chapter 6. Model Generation Examples

6.1. Introduction

In this chapter several examples of model generation from conceptual graphs are presented. The conceptual graphs have been taken from the test suite created for the Model Generator. First an example is explained in detail. The rest of the examples are then presented.

6.2. A Detailed Example

A detailed analysis of the first example is now presented. For the sentence given below the conceptual graph is shown in Figure 6.1.

" the processor begins the execution of the program". (3)

The conceptual graph is generated after parsing and semantically analyzing the sentence.

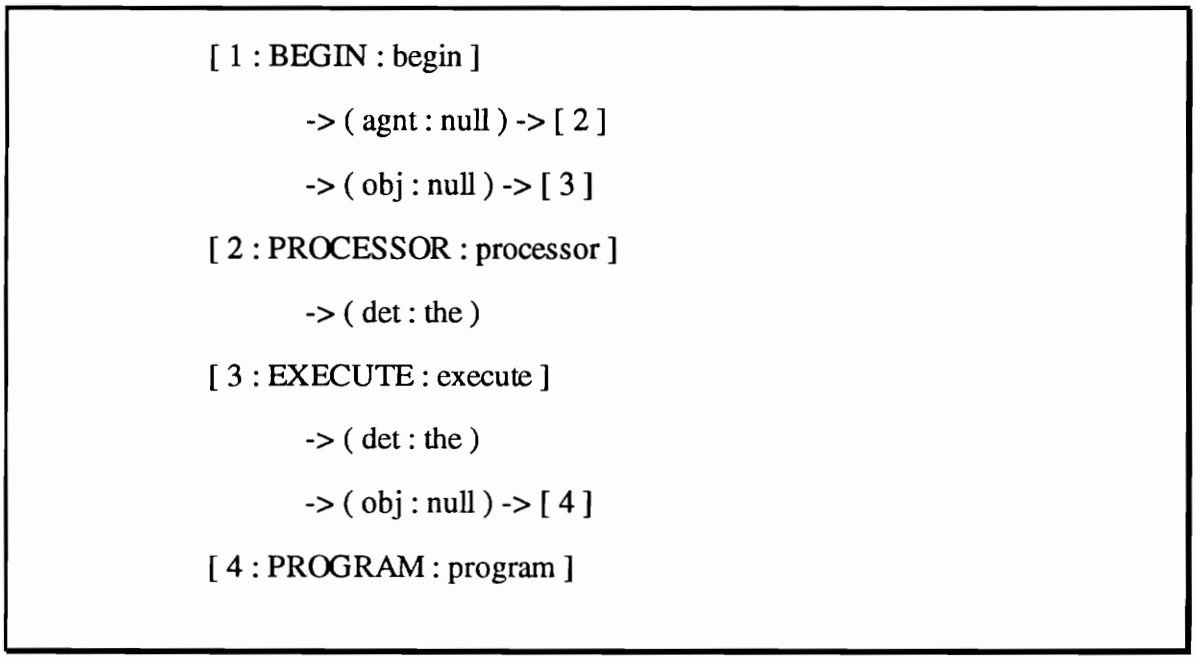


Figure 6.1. Conceptual graph for Sentence #3

The main concepts in the conceptual graph are : PROCESSOR, BEGIN, EXECUTE and PROGRAM. Table 3 presents these concepts along with their supertypes.

Table 3. Concept Types and their Supertypes in Sentence #3.

CONCEPT	TYPE
PROCESSOR	DEVICE
BEGIN	EVENT
EXECUTE	ACTION
PROGRAM	VALUE

Figure 6.2 shows the canonical graphs and canonical pictures for the four words.

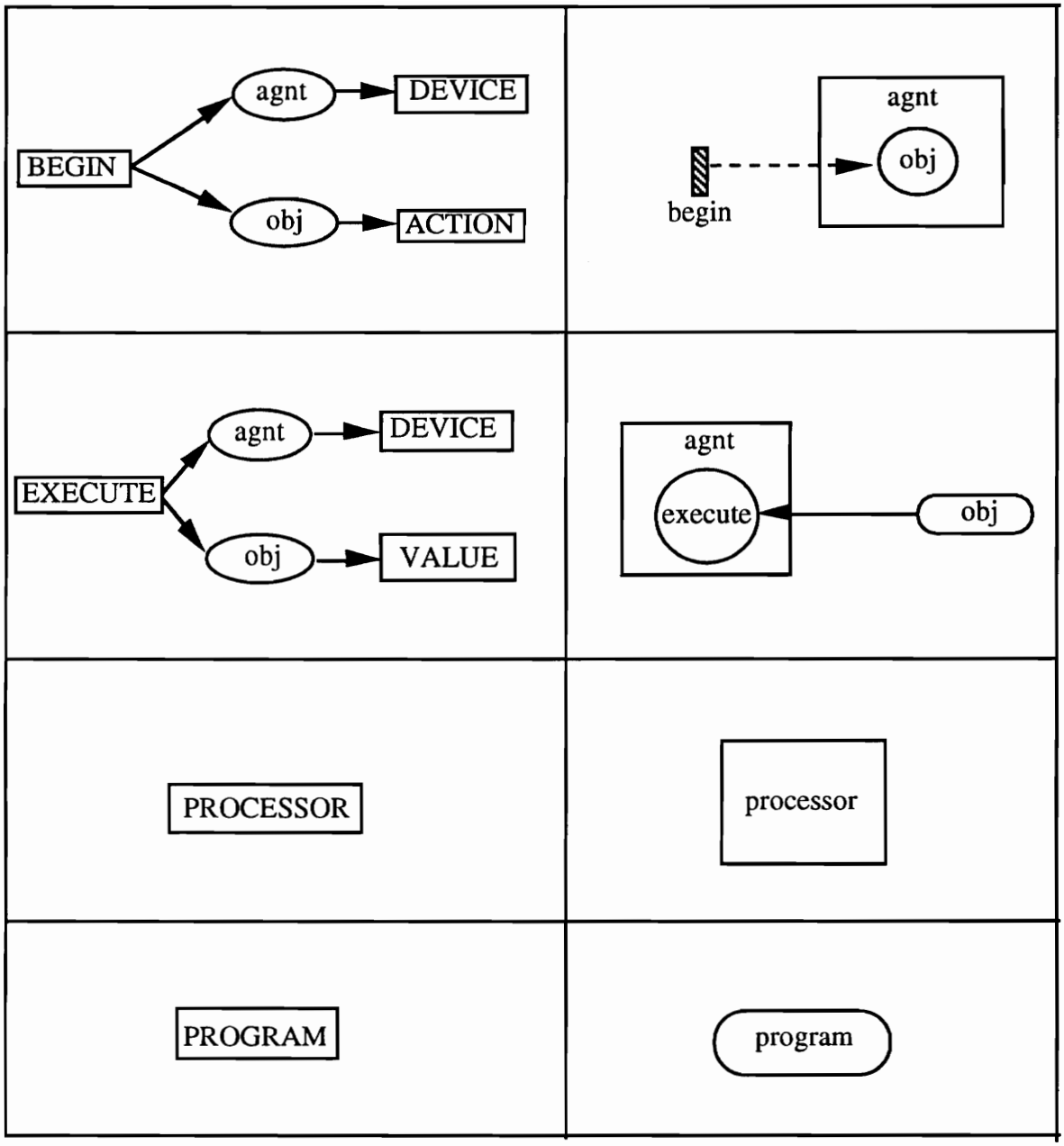


Figure 6.2. Canonical Graphs and Canonical Pictures for words in Sentence #3.

Since BEGIN is the first concept in the conceptual graph, its canonical picture script is retreived first from the Interpretatin Library. The script is shown in Figure 6.3.

```
[1] trans_bar ( BEGIN , 225 , 325 , length , height , { } )
[2] rect ( agnt , 200 , 300 , length , height , { } )
[3] circle ( obj , 350 , 330 , length , height , { } )
[4] arrow ( name , from 2, to 1 , 0 , 0 , { } )
[5] arrow ( name , from 1, to 3 , 0 , 0 , { } )
```

Figure 6.3. Script for Canonical Picture for BEGIN.

The generic labels in the canonical picture script are then instantiated by the Mapper. This involves replacing the relation type labels by the related concept node information provided in the input conceptual graph. This is shown in Figure 6.4.

```
[1] trans_bar ( begin : 1 , 225 , 325 , length , height , { } )
[2] rect ( processor : 2 , 200 , 300 , length , height , { } )
[3] circle ( execute : 3 , 350 , 330 , length , height , { } )
[4] arrow ( name , from 2, to 1 , 0 , 0 , { } )
[5] arrow ( name , from 1, to 3 , 0 , 0 , { } )
```

Figure 6.4. Script for BEGIN with instantiated labels.

Then the canonical picture script for PROCESSOR is retrieved and the generic labels replaced by the node information from the conceptual graph. The script is shown below.

```
[1] rect ( processor : 2 , 200 , 300 , length , height , { } )
```

Figure 6.5. Script for PROCESSOR with instantiated label.

This is then appended to the script in Figure 6.4. Multiple references are removed from the resulting script and the command index numbers are updated. This results in the same script as in Figure 6.4.

After this the script for EXECUTE is retrieved and its generic labels instantiated. This is shown in Figure 6.6.

```
[1] circle ( execute : 3 , 225 , 325 , length , height , { } )
[2] rect ( program : 4 , 150 , 335 , length , height , { } )
[3] arrow ( name , from 1, to 2 , 0 , 0 , {} )
```

Figure 6.6. Script for EXECUTE with instantiated nodes.

This is appended to the script in Figure 6.4. The redundancies are removed and connectives redirected. Figure 6.7 shows the resulting intermediate script.

```
[1] trans_bar ( begin : 1 , 225 , 325 , length , height , {} )
[2] rect ( processor : 2 , 200 , 300 , length , height , { } )
[3] circle ( execute : 3 , 350 , 330 , length , height , { } )
[4] arrow ( name , from 2, to 1 , 0 , 0 , {} )
[5] arrow ( name , from 1, to 3 , 0 , 0 , {} )
[6] rect ( program : 4 , 150 , 335 , length , height , { } )
[7] arrow ( name , from 3, to 6 , 0 , 0 , {} )
```

Figure 6.7. Intermediate Script.

Finally, the script for the canonical picture for PROGRAM is retrieved, its generic nodes instantiated, the script appended to the earlier assembled script and redunancies removed.

The script for representing the complete sentence is is the same as in Figure 6.7. The script in Figure 6.7 also shows the default display coordinates to be used by the Placer. The script is used by the Placer to generate display coordinates using the force-directed placement algorithm explained in Chapter 5. Figure 6.8 is the screen image produced by the script before the placement is done, and Figure 6.9 shows the final picture displayed.

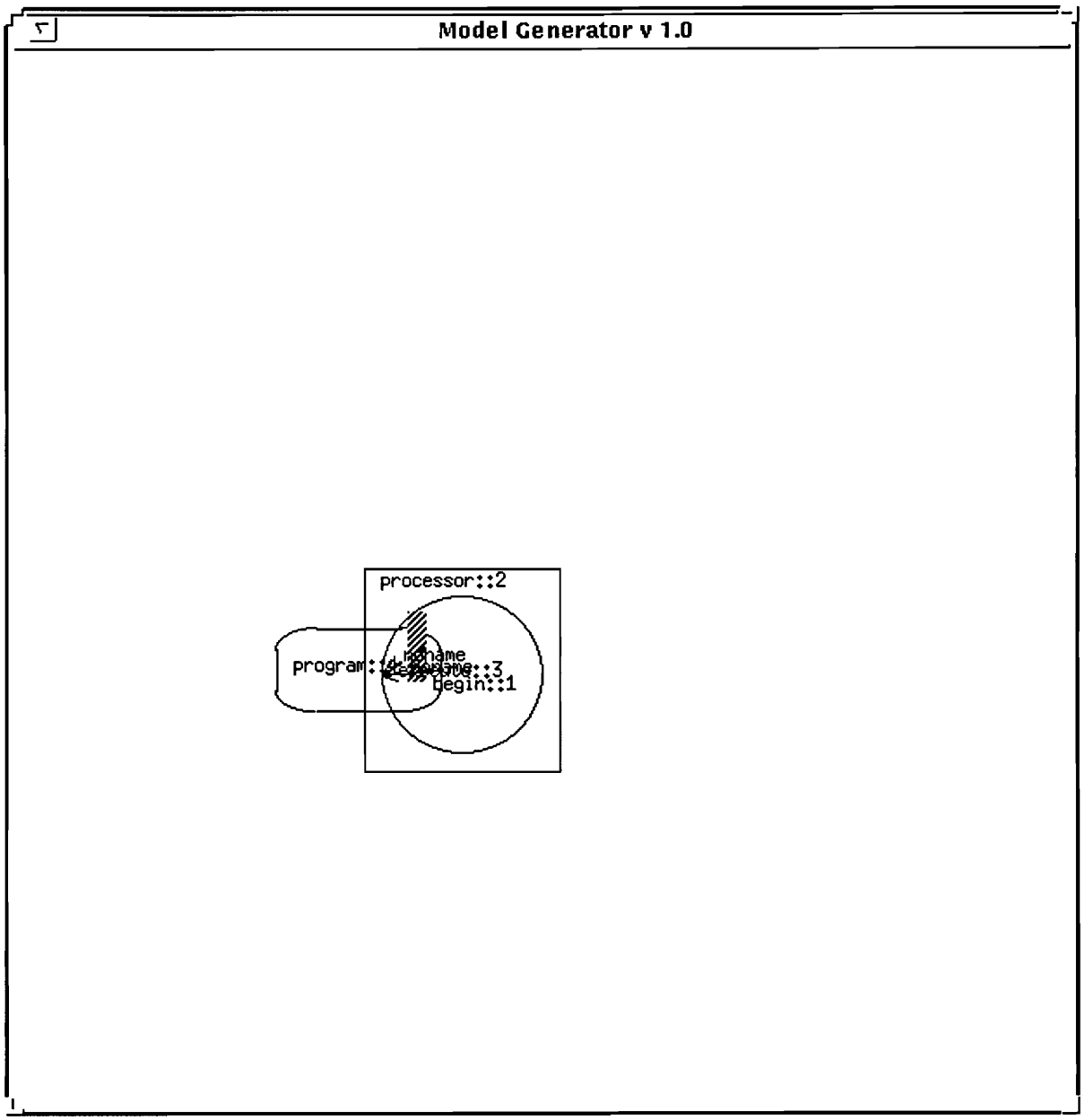


Figure 6.8. Graphical Model for Sentence #3 before Placement.

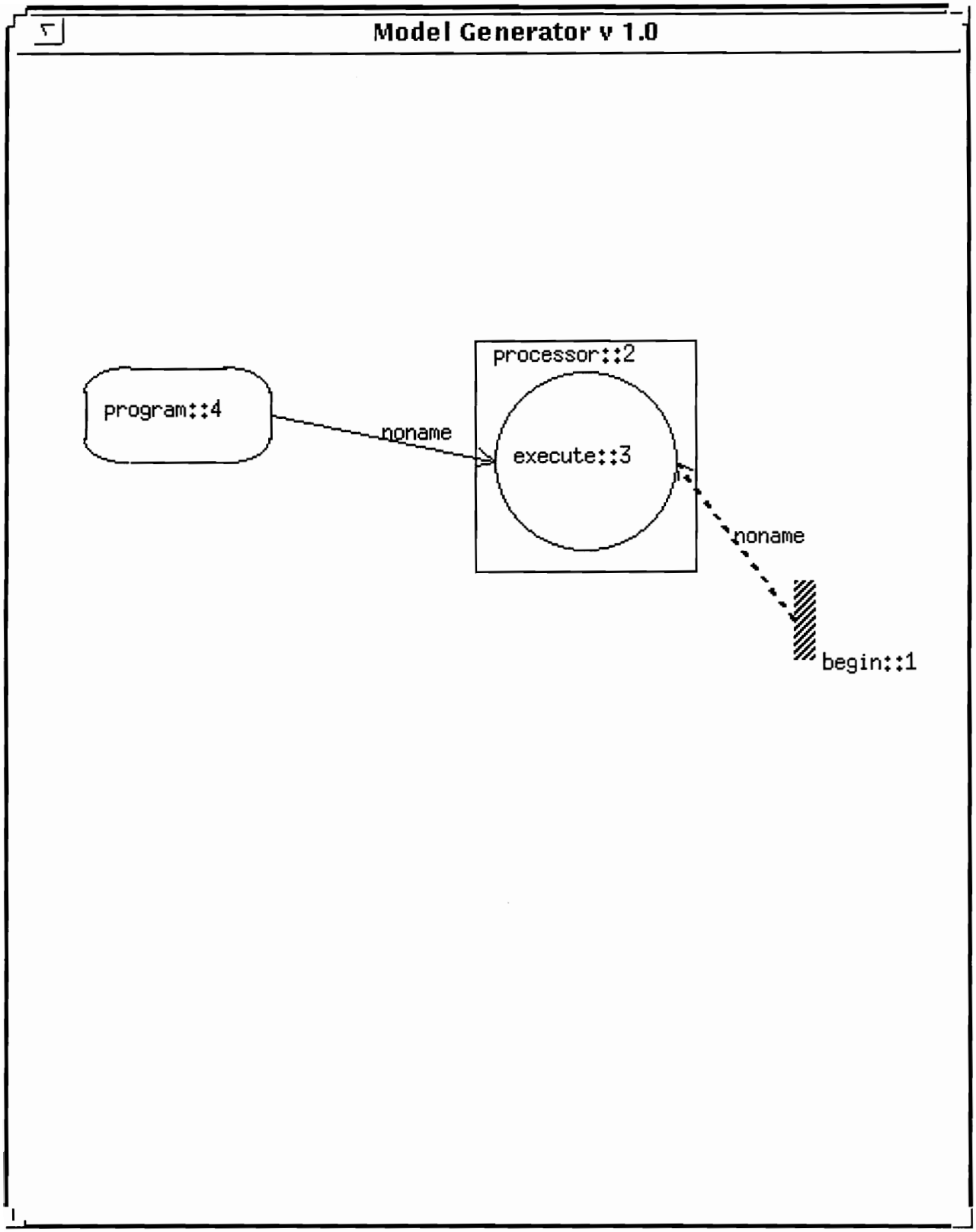


Figure 6.9. Graphical Model for Sentence #3 after Placement.

6.3. Other Model Generation Examples

This section contains four more examples of graphical models created from conceptual graphs for sentences taken from the suite of conceptual graphs. These examples highlight the capabilities of the Model Generator.

The first example presented in this section is for the sentence given below.

" the i80486 contains a cache, processor and co-processor." (4)

The conceptual graph for sentence #4 is shown in Figure 6.10.

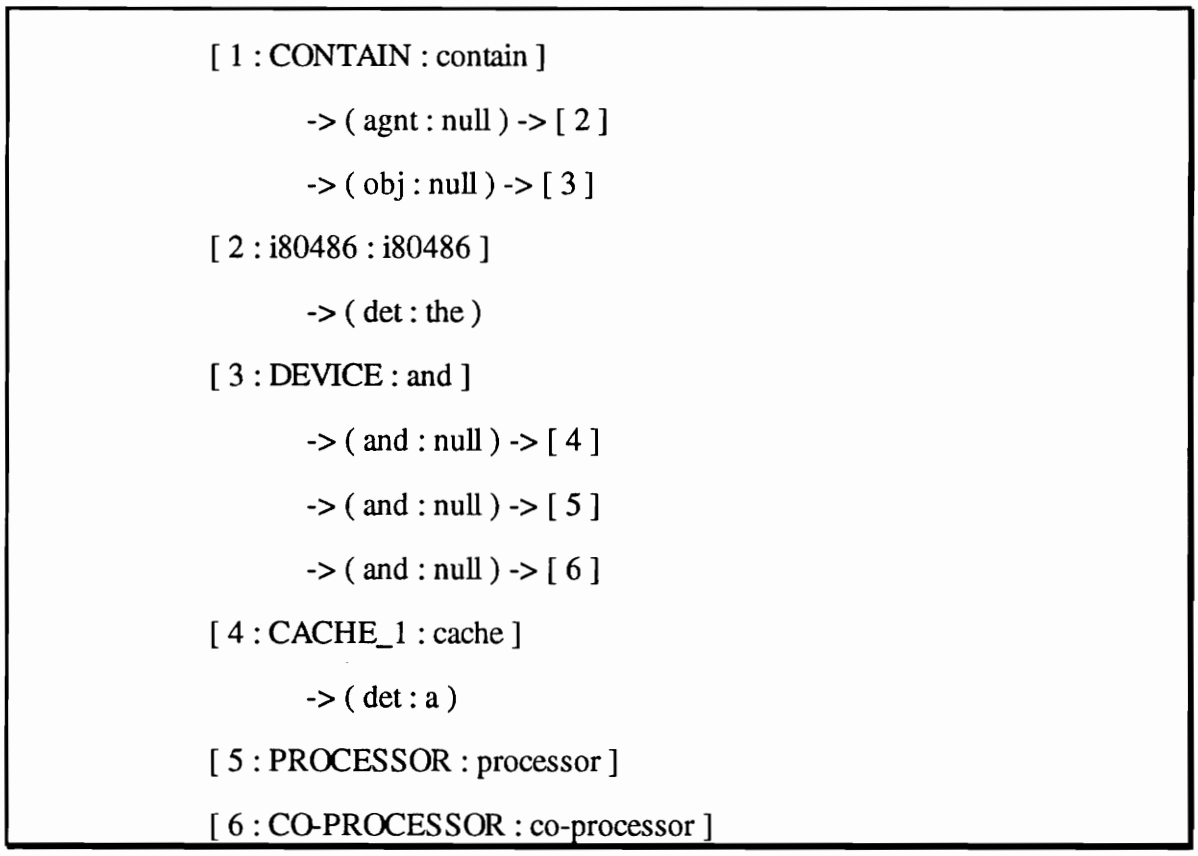


Figure 6.10. Conceptual graph for Sentence #4.

The above sentence is a structural specification and the block diagram notation is used to graphically represent it. The main concepts in the conceptual graph are: CONTAIN, i80486, CACHE, PROCESSOR and CO-PROCESSOR. Also present in the conceptual graph is the conjunction AND indicating parts of the abstract concept DEVICE. The final script generated by the Mapper from the above conceptual graph is shown in Figure 6.11(a).

```
[1] rect ( i80486 : 2 , 225 , 325 , length , height , { 2, 3, 4 } )
[2] rect ( cache_1 : cache : 4 , 450 , 450 , length , height , { } )
[3] rect ( processor : 5 , 450 , 450 , length , height , { } )
[4] rect ( co-processor : 6 , 450 , 450 , length , height , { } )
```

Figure 6.11(a). Script for Sentence #4 with default settings.

The figure above shows default display coordinates. The Placer recomputes the coordinates and sizes before displaying the final representation. Figure 6.11(b) shows the script with revised coordinates. The graphical model generated is shown in Figure 6.12.

```
[1] rect ( i80486 : 2 , 300 , 400 , 420 , 90 , { 2, 3, 4 } )
[2] rect ( cache_1 : cache : 4 , 320 , 425 , 130 , 50 , { } )
[3] rect ( processor : 5 , 460 , 425 , 100 , 50 , { } )
[4] rect ( co-processor : 6 , 574 , 425 , 124 , 50 , { } )
```

Figure 6.11(b). Script for Sentence #4 after final placement.

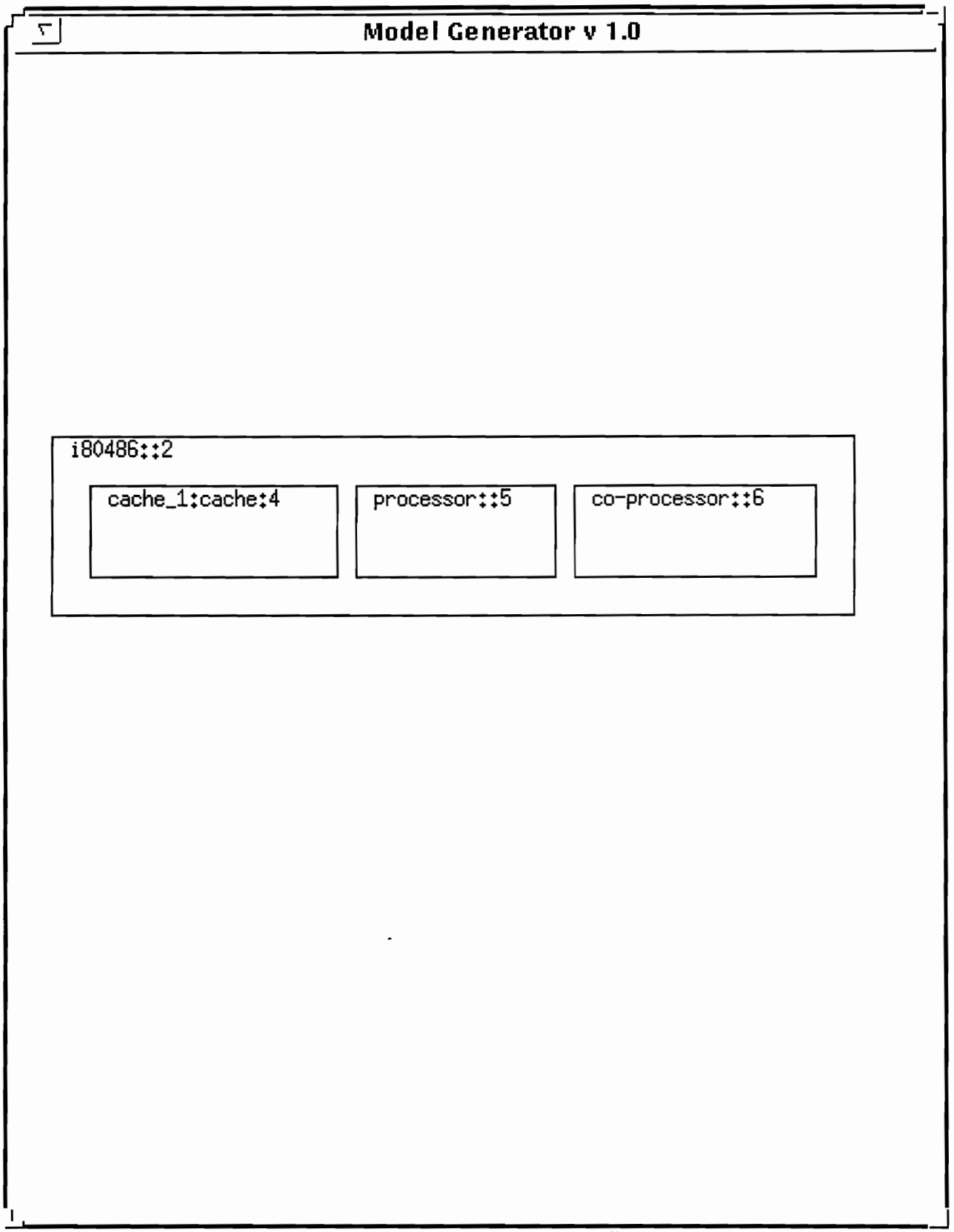


Figure 6.12. Graphical Model for Sentence #4 after Placement.

The second example of model generation is for the following sentence.

" resetting, initializing and loading clear the memory . " (5)

The conceptual for this sentence after parsing and semantic analysis is shown below.

```
[ 1 : CLEAR : clear ]
        -> ( agnt : null ) -> [ 2 ]
        -> ( obj : null ) -> [ 3 ]
[ 2 : WRITE : and ]
        -> ( and : null ) -> [ 4 ]
        -> ( and : null ) -> [ 5 ]
        -> ( and : null ) -> [ 6 ]
[ 4 : RESET : reset ]
[ 5 : INITIALIZE : initialize ]
[ 6 : LOAD : load ]
[ 3 : VALUE : * ]
        -> ( loc : null ) -> [ 7 ]
[ 7 : MEMORY : memory ]
        -> ( det : the )
```

Figure 6.13. Conceptual graph for Sentence #5.

This conceptual graph has the conjunction AND, but unlike in the previous example, it indicates subparts that are ACTIONS. RESET, INITIALIZE and LOAD are all subtypes of the action WRITE in the conceptual type hierarchy of the system. The action WRITE causes the CLEAR of the MEMORY. Any write to a memory actually means a write to the

value in the memory and this is inidicated by the generic concept VALUE which is *located* in MEMORY.

The unplaced script for the graphical representation for sentence #5 is shown in the figure below.

```
[1] circle ( clear : 1 , 350 , 325 , length , height , { } )
[2] roundrect ( value : 3 , 425 , 340 , length , height , { } )
[3] circle ( write : and : 2 , 225 , 450 , length , height , { 8 , 9 , 10 } )
[4] cause_bar ( cause , 450 , 450 , length , height , { } )
[5] arrow ( name , from 3, to 4 , 0 , 0 , {} )
[6] arrow ( name , from 4, to 1 , 0 , 0 , {} )
[7] arrow ( name , from 1, to 2 , 0 , 0 , {} )
[8] circle ( reset : 4 , 450 , 450 , length , height , { } )
[9] circle ( initialize : 5 , 450 , 450 , length , height , { } )
[10] circle ( load : 6 , 450 , 450 , length , height , { } )
[11] rect ( memory : 7 , 450 , 450 , length , height , { 2 } )
```

Figure 6.14. Script for Sentence #5.

The generated model is shown in Figure 6.15.

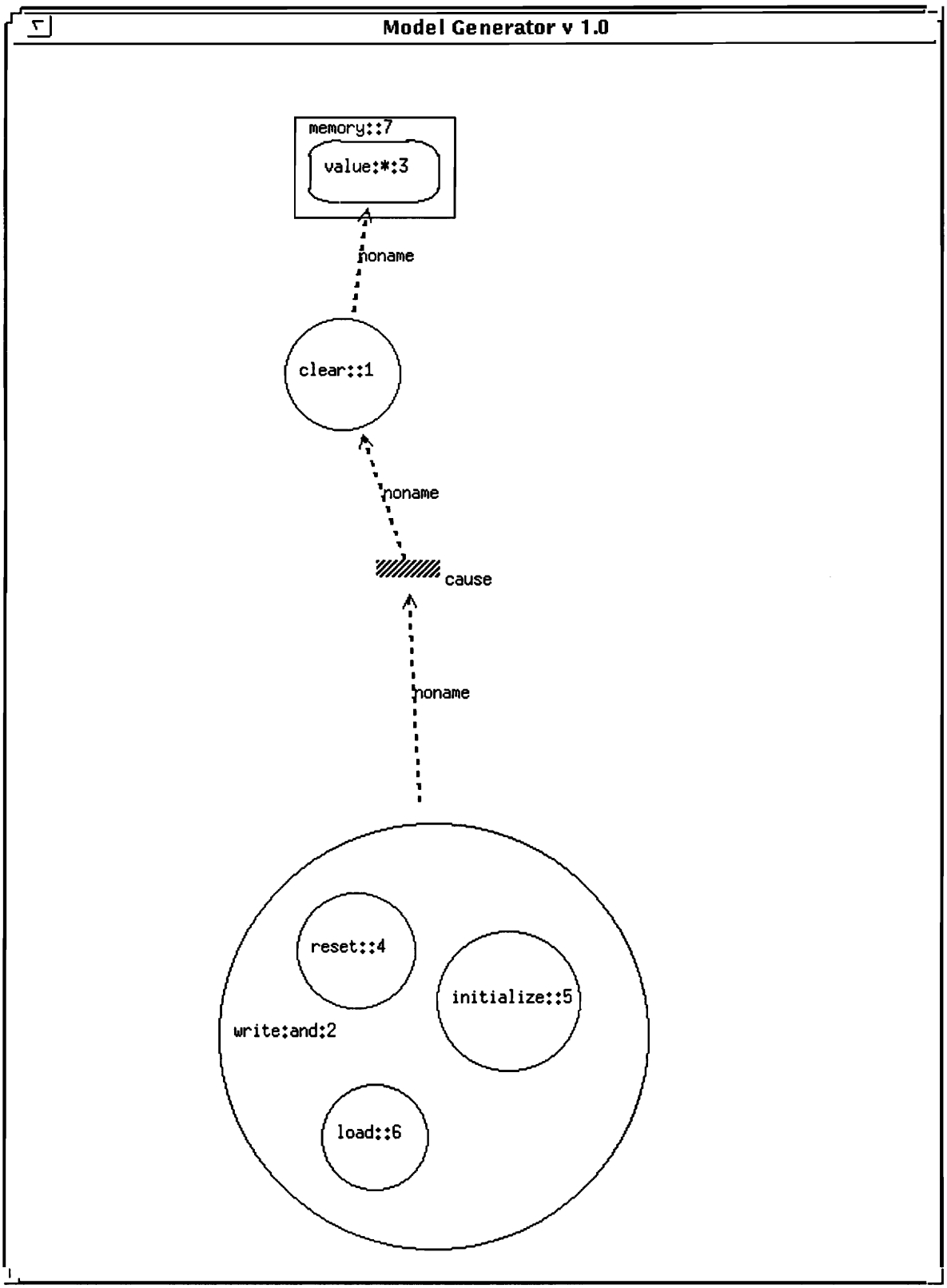


Figure 6.15. Graphical Model for Sentence #5 after Placement.

The third example presented in this section is the conceptual graph for the sentence given below.

" execution of the OUT instruction connects the peripheral device to the processor on the 16-bit bus ." (6)

The conceptual graph for this sentence is shown in Figure 6.16.

```
[ 1 : EXECUTE : execute ]
        -> ( det : the )
        -> ( obj : null ) -> [ 3 ]
[ 2 : CONNECT : connect ]
        -> ( obj : null ) -> [ 4 ]
        -> ( dest : null ) -> [ 5 ]
        -> ( inst : null ) -> [ 6 ]
        -> ( agnt : null ) -> [ 1 ]
[ 3 : INSTRUCTION : instruction ]
        -> ( name : null ) -> [ 7 ]
[ 4 : PROCESSOR : processor ]
[ 5 : DEVICE : peripheral ]
[ 6 : CARRIER : bus ]
        -> ( type : adj ) -> [ 8 ]
[ 7 : ID : OUT ]
[ 8 : MEM_MEASURE : 16-bit ]
```

Figure 6.16. Conceptual graph for Sentence #6.

The unplaced script for the conceptual graph is shown below in Figure 6.17.

```
[1] circle ( execute : 1 , 225 , 325 , length , height , { } )
[2] roundrect ( instruction : OUT : 3 , 150 , 335 , length , height , { } )
[3] solarrow ( name , from 2, to 1 , 0 , 0 , {} )
[4]  circle ( connect : 2 , 350 , 325 , length , height , { } )
[5] rect ( processor : 4 , 125 , 275 , length , height , { } )
[6] rect ( device : peripheral : 5 , 325 , 275 , length , height , { } )
[7] cause_bar ( cause , 310 , 335 , 30 , 20 , { } )
[8] solidline ( carrier : bus : 16-bit : 6 , from 5, to 6 , 0 , 0 , { 4 } )
[9] arrow ( name , from 1, to 8 , 0 , 0 , {} )
[10] arrow ( name , from 8, to 4 , 0 , 0 , {} )
```

Figure 6.17. Script for Sentence #6.

The graphical model generated is shown in Figure 6.18.

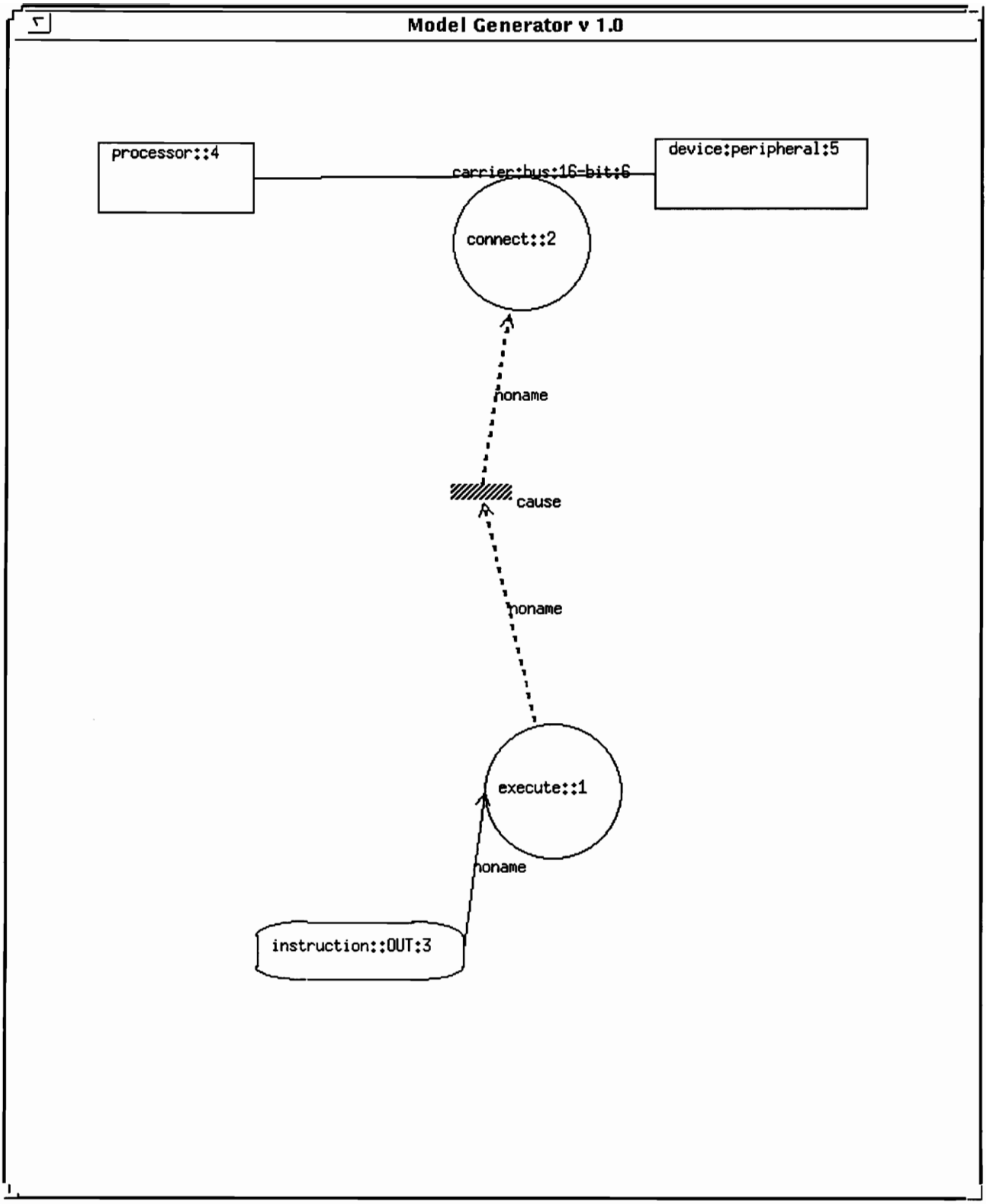


Figure 6.18. Graphical Model for Sentence #6 after Placement.

The last example presented in this section is the conceptual graph for the sentence given below.

"the processor accesses the memory M1 when chip-select is high". (7)

The conceptual graph for this sentence is shown in Figure 6.19.

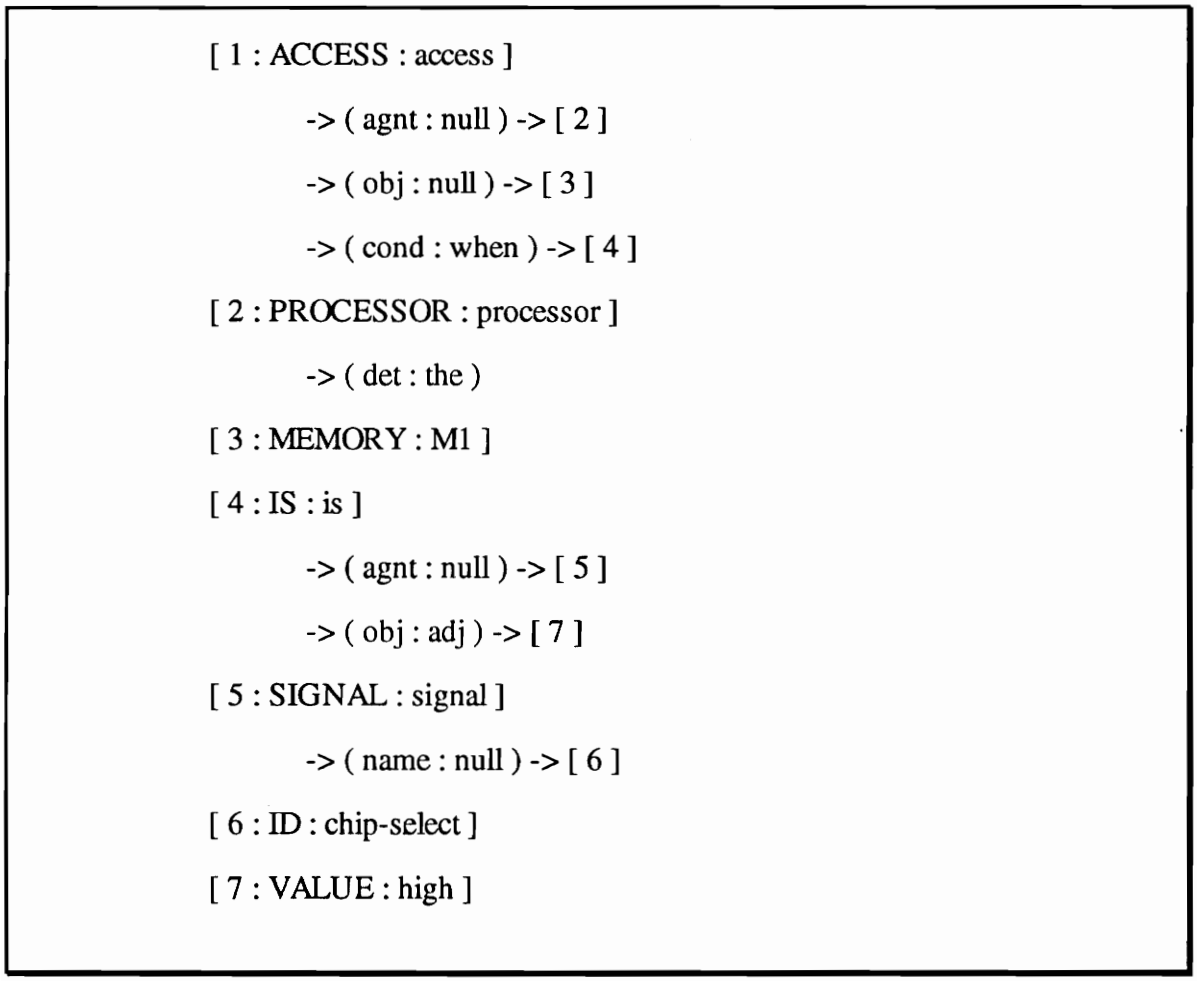


Figure 6.19. Conceptual graph for Sentence #7.

In the conceptual graph the concept IS indicates that a high VALUE of SIGNAL chip-select is a condition for the ACCESS. This basically means that the STATE 'IS' equates the value on chip-select with a high value to form the condition. This is highlighted by using an "=" sign instead of IS in the graphical representation. The unplaced script for the conceptual graph is shown below in Figure 6.20.

```
[1] circle ( access : 1 , 225 , 325 , length , height , { } )
[2] rect ( processor : 2 , 200 , 300 , length , height , { } )
[3] rect ( memory : M1 : 3 , 350 , 300 , length , height , { } )
[4] arrow ( name , from 1, to 3 , 0 , 0 , {} )
[5] circle ( is : 4 , 335 , 125 , length , height , { } )
[6] trans_bar ( cond , 210 , 135 , 30 , 20 , { } )
[7] arrow ( name , from 5, to 6 , 0 , 0 , {} )
[8] arrow ( name , from 6, to 1 , 0 , 0 , {} )
[9] roundrect ( signal : chip-select : 5 , 225, 340 , length , height , { } )
[10] roundrect ( value : high : 7 , 425, 340 , length , height , { } )
[11] arrow ( name , from 10, to 5 , 0 , 0 , {} )
[12] arrow ( name , from 11, to 5 , 0 , 0 , {} )
```

Figure 6.20. Script for Sentence #7.

The graphical model generated is shown in Figure 6.21.

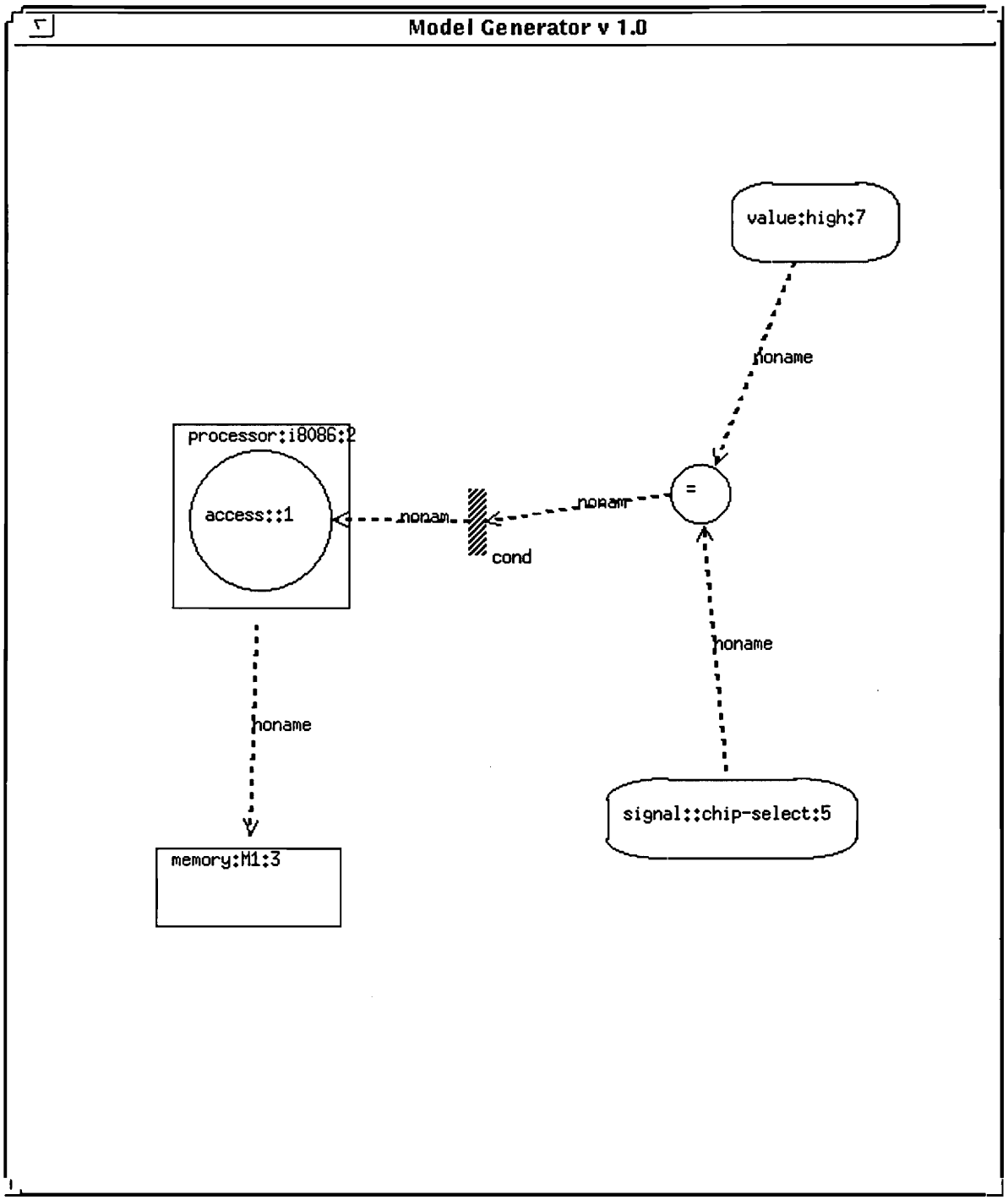


Figure 6.21. Graphical Model for Sentence #7 after Placement.

Chapter 7. Future Work

7.1. Back Annotation

The ASPIN system generates formal models from informal specifications. To obtain validation of English specification sentences, each sentence is parsed and semantically analyzed to map it into a conceptual graph. This conceptual graph is then used by the Model Generator to display a graphical model, which the specification author can validate.

If the author finds the interpretation of the sentence, by the system to be incorrect, he has to rewrite the sentence and re-enter it in the system. This undermines the ASPIN system objective of rapid validation and modeling of system specifications. The usefulness of the system would be greatly enhanced if the author could correct the graphical model and obtain a corrected conceptual graph and automatically rewritten sentence. This is known as *back-annotation*. After the correct conceptual graph has been generated, it can be used to create formal VHDL models as well as generate corrected specification sentences.

The X-Windows interface of the Model Generator can be utilized for back-annotation purpose. The user can *select, move, delete* or *add* icons and connectives using mouse inputs. These actions can then be reflected in a modified conceptual graph.

Moving an icon, for example, from inside another icon, shows change in containments. This can be reflected by the deletion of the relation *loc* (location) of the concept represented by the moved icon. If an icon is moved inside another icon, then the *loc* relation needs to be added to the conceptual graph. If a subtype concept icon is moved inside an icon, then conjunction needs to be added to the conceptual graph. If an icon is *deleted* form the graphical model, then the corresponding concept and all its relations should be deleted from the original conceptual graph. Similarly, *adding* an icon to the picture should add a new concept to the original conceptual graph. Connectives between the new icon and older ones will define the relationships of the new concept in the conceptual graph. A facility for editing icon labels also needs to be provided. A changed label of an icon can be reflected as a changed referent field for the corresponding concept. Editing labels would also require resizing of icons to accomodate the change in label size.

To achieve the back-annotation capability the *drawtype* data structures for all selected icons will have to be identified. The position of the mouse cursor can select icons. A data base, that correlates the sequence of mouse inputs with the information in the data structure of the selected icons, can then be used to modify the conceptual graph.

Once the conceptual graph has been modified, a system that converts conceptual graphs to English sentences can be used to rewrite the specification sentence.

7.2. Navigational Control

Currently, the ASPIN system handles single sentences at a time. As multi-sentence handling capability is added, the number and complexity of the icons in the graphical model will increase. This will make a quick validation difficult for the user. To aid the user, the system should be able to 'zoom out' and 'zoom in' in the graphical model. Zooming in an icon would cause the icon to occupy the whole screen, facilitating the user to observe all the details of that icon. Similarly, zooming out would cause the devices and data flows, or actions and control dependencies to be hidden, allowing the user to view the representation from a higher level of abstraction.

7.3. Improvements

For the Model Generator to be more effective, the vocabulary of the system needs to be increased. This would require encoding canonical pictures in the Interpretation Library for the increased vocabulary. Another improvement possible for the Model Generator can be in the placement of icons. Better heuristics can be developed to minimize crossing of connectives with other connectives and icons. This would require further experimentation with the existing algorithm to determine optimum constants used in the functions employed.

Bibliography

[1] W. Cyre and J. Armstrong, "System Level Design Automation: Engineering Models from Informal Specifications," Research Report, Virginia Polytechnic & State University, 1992.

[2] W. R. Cyre, "Towards Synthesis from English Descriptions," Proc. 26th Design Automation Conference, Las Vegas, NV, June 1989.

[3] W.R. Cyre. "Integrating Specification requirements for Automated Interpretation," Second Intl. Workshop on Rapid System Prototyping, 1991.

[4] W.R. Cyre. "Design of a Restricted Sublanguage," Proc. Theoretical Approaches to Natural Languages Understanding Workshop, Halifax, Novia Scotia, May 28- 30,1985.

[5] P. Eades, "A Heuristic for Graph Drawing," Congressus Numerantium, 42, pp. 149-160, 1984.

[6] J. Fargues, et al., "Conceptual Graphs for Semantics and Knowledge Processing," IBM J. Res. Development, Vol. 30, No. 1, January 1986.

[7] T. Fruchterman and E. Reingold, "Graph Drawing by Force-deirected Placement," Software-Practice and Experience, Vol. 21(11), November 1991.

[8] R. Greenwood, "Semantic Analysis for System Level Design Automation," Master's Thesis, Virginia Polytechnic & State University, Blacksburg, Virginia, 1992.

[9] Jack Gregg, "Block Diagrams Simplify Mechanism Models," Machine Design", Jan 25, 1990.

[10] Kar-Wing E. Lor, "Operational Definitions for System Requirements as the Basis of Design Automation" Software - Practice and Experience, Vol. 21(10), p. 1103-1124, John Wiley & Sons, Ltd., 1991.

[11] J.L. Peterson, "Petri Nets," Computing Surveys, Vol.9, No. 3, September 1977.

[12] L.B. Protsko, et al., "Towards the Automatic Generation of Software Diagrams," IEEE Trans. on Software Engineering, Vol.17, No. 1, January 1991.

[13] N. Quinn, "The Placement Problem as viewed from the Physics of Classical Mechanics," General Dynamics Corporation.

[14] N. Quinn and M. Breur, "A force directed component placement procedure for printed circuit boards," IEEE Trans. on Circuits and Systems, CAS-26, (6), pp. 377-388, 1979.

[15] W. Reynolds, The Block Diagram as a Mathematical Model and a Computer Language," Proc. 11 th. Annual Advanced Control Conference, 1985.

[16] B. Singh, et al., "The Modeler's Assistant: A CAD Tool for Behavioral Model Development," CHDL '93.

[17] J.F. Sowa, Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley, Reading, MA, 1984.

[18] J.F. Sowa and E.C. Way, "Implementing a Semantic Interpreter Using Conceptual Graphs," IBM J. Res. Development, Vol. 30, pp. 57-69, January 1986.

[19] H. A. Spang, et. al., "An Evaluation of Block Diagram CAE Tools, " Proc. 11 th. Triennial World Congress of the Internal Federation of Automatic Control, Tallinn, USSR, Aug. 1990.

[20] R Tammassia, et al., "Automatic Graph Drawing and Readability of Diagrams," IEEE Trans. Syst., Man., Cybern., Vol. 18, No. 1, pp. 61-79, Jan./Feb. 1988.

[21] P. A. Tipler, "Physics," 2 nd. Edition , Worth Publishers, New York, 1982.

[22] Petri, C. A., "Concepts of Net Theory," Proc. Symp. and Summer School on Mathematical Foundations of Computer Science, High Tatras, pp. 137-146, Sept. 3-8, 1973.

[23] Holt, A.W. and Commoner, F, "Events and Conditions," Applied Data Research, New York, 1970.

[24] "Xlib Programming Manual", O'Reilly & Associates, Inc.

[25] "Xlib Reference Manual", O'Reilly & Associates, Inc.

Appendix A. User's Manual

This appendix includes the User's Manual for the Model Generator. The Model Generator software is written in the C programming language under the X-Windows environment on the Sun Sparcstation platform. First the operating instructions are provided. Then the file organization of the Model Generator is explained. Then the instructions for updating the database are outlined.

Operating Instructions

The Model Generator can be invoked by the command,

modgen

On entering this command from the UNIX shell, the Model Generator menu appears on the screen. The menu format is shown below.

0. quit

1. from file

2. from mgsuite.log

your option:

The input conceptual graph can either be in an external file or can be selected from the suite file (mgsuite.log) created for testing the Model Generator.

If *option 1* is selected, the user is prompted to provide the name of a file.

enter filename:

The file name entered should contain the single conceptual graph for which the graphical model is desired. If *option 2* is selected, the following prompt appears,

automatic (y/n) :

If a "y" is entered, then the complete suite is displayed one by one. Pressing the "F1" key by keeping the cursor on the display window generates the graphical model for the next conceptual graph in the suite. At the end of the file, the programs returns to the main menu. The conceptual graphs are numbered 1-60 in the suite. The corresponding specification sentences are also provided.

If the user enters a "n", then the user is prompted to enter the sentence number.

enter sentence number:

Only the graphical representation for that sentence is then displayed. The session can be terminated by choosing *option 0* in the main menu.

Program *modgen* is the shell which invokes the main program after obtaining the user option. The user can also run the Model Generator by simply typing,

generate *filename*

The file containing the input conceptual graph is specified by *filename*.

Filesystem Organization

The main program in the Model Generator resides in a file called *generate.c*. The *main()* routine in this file calls all the major routines for model generation. This routine also interacts with the X-Window interface routines.

The X-Window interface routines are found in following files.

initx.c : Initializes connection with the X server.
windowx.c : Sets up the display, window and window manager parameters.
drawx.c : Contains drawing routines using X library functions.
emudrawx.c : Contains drawing routines using Emu library functions.
eventx.c : Event handlers for the X-Windows client-server connection.

textx.c : Text handling routines.

quitx.c : Breaks connection with the X server.

The other files used by the Model Generator are given below.

generate.c

extract.c

interp.c

dbase.c

base.c

allocate1.c

miscl.c

File *extract.c* contains routines to read the conceptual graph information into internal data structures. It also contains routines for script assembly and manipulation. The routines needed to generate icons from the assembled script are contained in file *interp.c*. Routines used to retrieve data base entries are in the file *dbase.c*. Sizing and dimension allocation is carried out by the routines found in the file *allocate1.c*. The force-directed placement routine is in the file *base.c*. Other miscellaneous utility routines are in the file *miscl.c*.

The source code can be compiled by using the "**make**" command from the UNIX shell. This uses the *makefile* to obtain the source files and compile options.

Database Management

The data base for the Model Generator consists of the Interpretation Library and the conceptual type hierarchy. The Interpretation Library is contained in the text file called *Intrp.lib*. The conceptual type hierarchy is contained in the file *cons.h*.

If a new concept needs to be added to the existing data base, it has to be added to the file *cons.h*. The script for its canonical graph should be added to the file *Intrp.lib*. It is not necessary to keep the concepts in alphabetical order, but is a recommended practice. The details of the Interpretation Library entries have been already explained. Care must be taken to ensure that all the fields in each script line be separated by single blanks in addition to the delimiters used.

Appendix B. Vocabulary

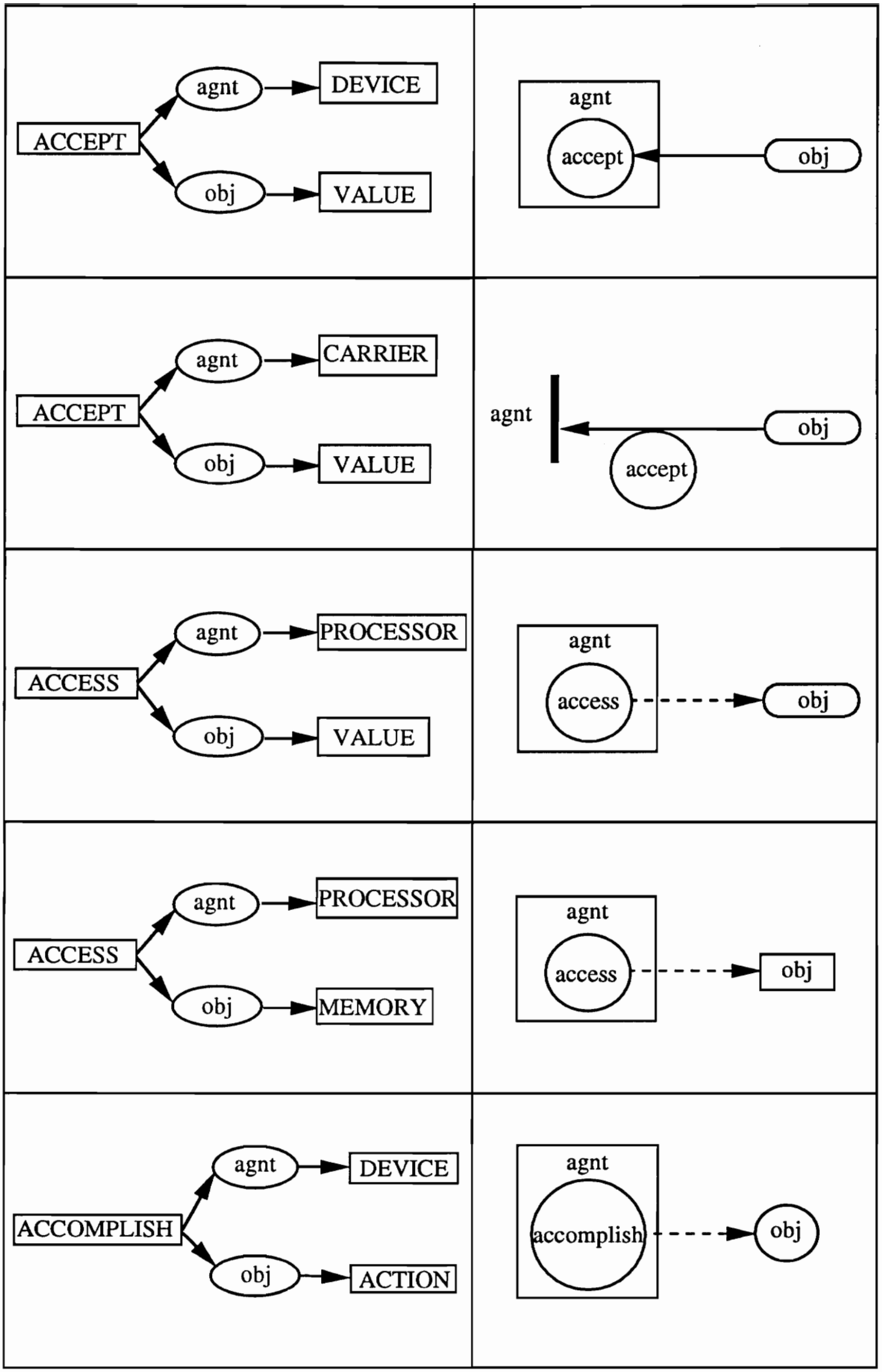
This appendix contains the list of words for which canonical pictures have been encoded in the Interpretation Library. The number in brackets indicates the number of different canonical pictures encoded for that word.

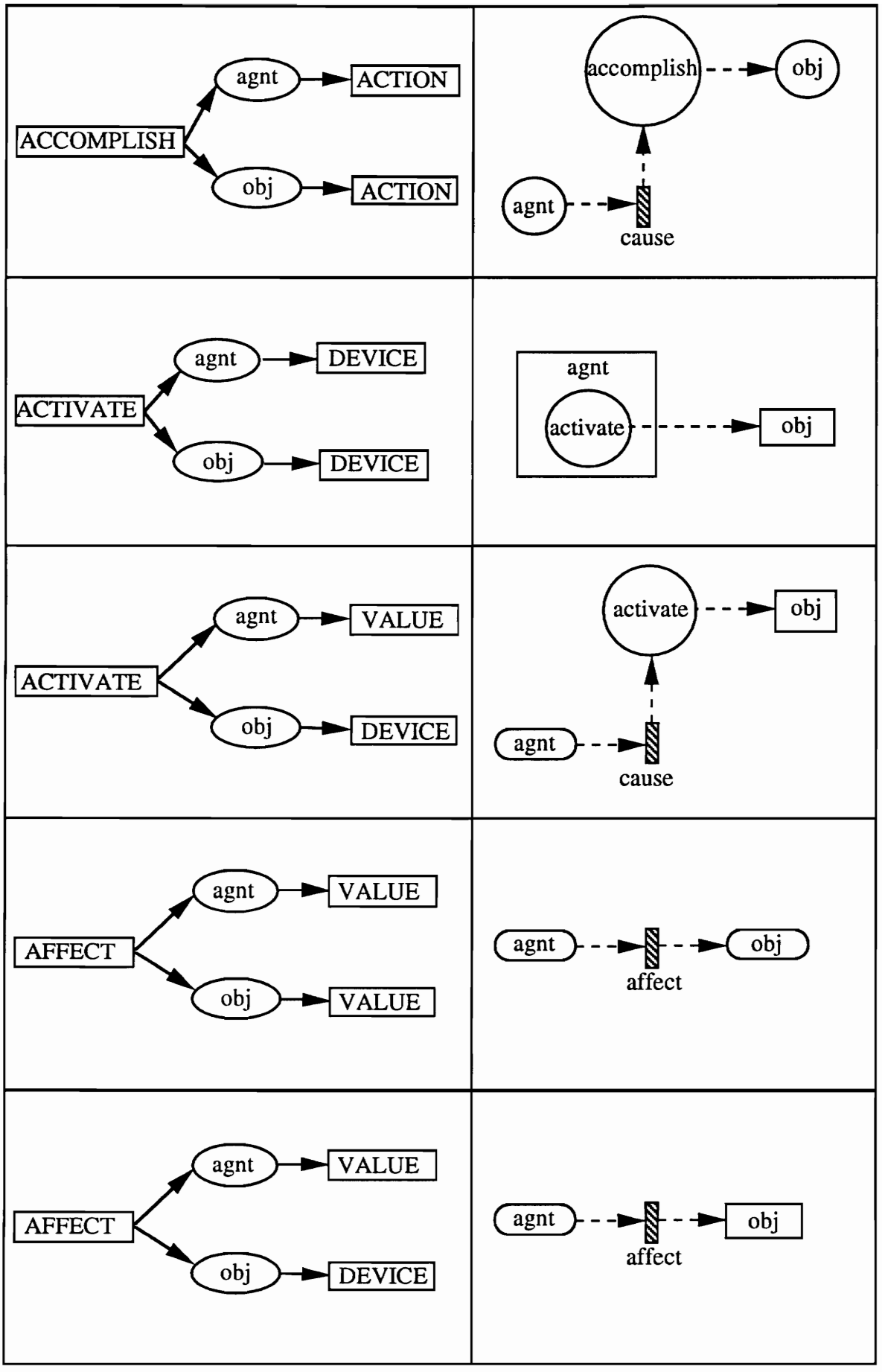
accept (2)
access (2)
accomplish (2)
activate (2)
affect (2)
begin
branch
cause
cache_1
clear (2)
connect
contain (2)
co-processor
fill
force (2)
generate
halt
have
increment (2)
indicate
initialize
input
instruction
is (2)
jump
latch
poll
processor
push
read
register
remove
represent
request
reset
rise
rotate
run
save

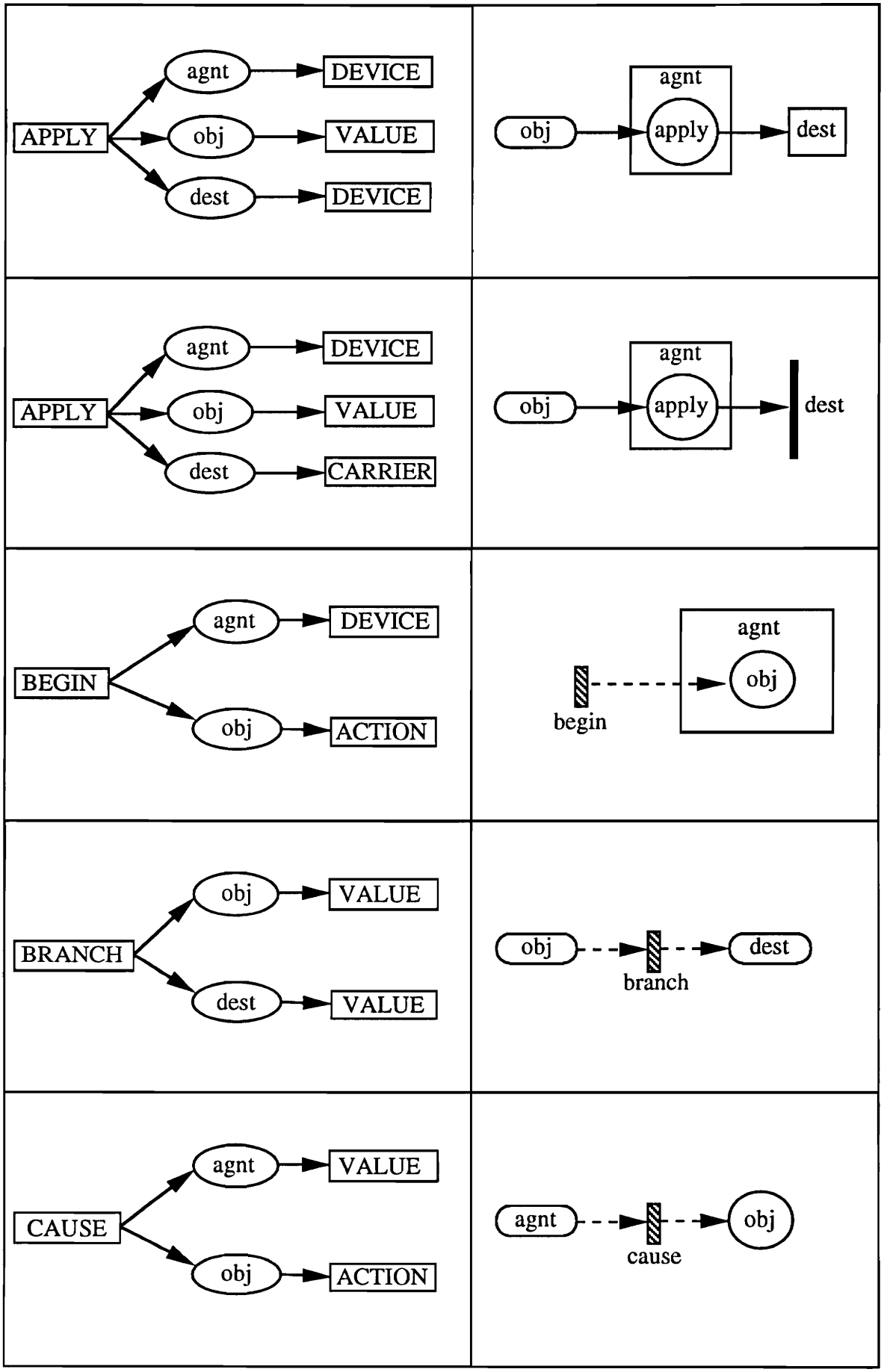
data
decode
delete
destroy
detect
disable
enable
enter
execute
fetch
load (2)
locate
manipulate
memory
modify
move
occupy
operate
output
perform
send
set
signal
specify
stop
store (2)
terminate
transfer
write

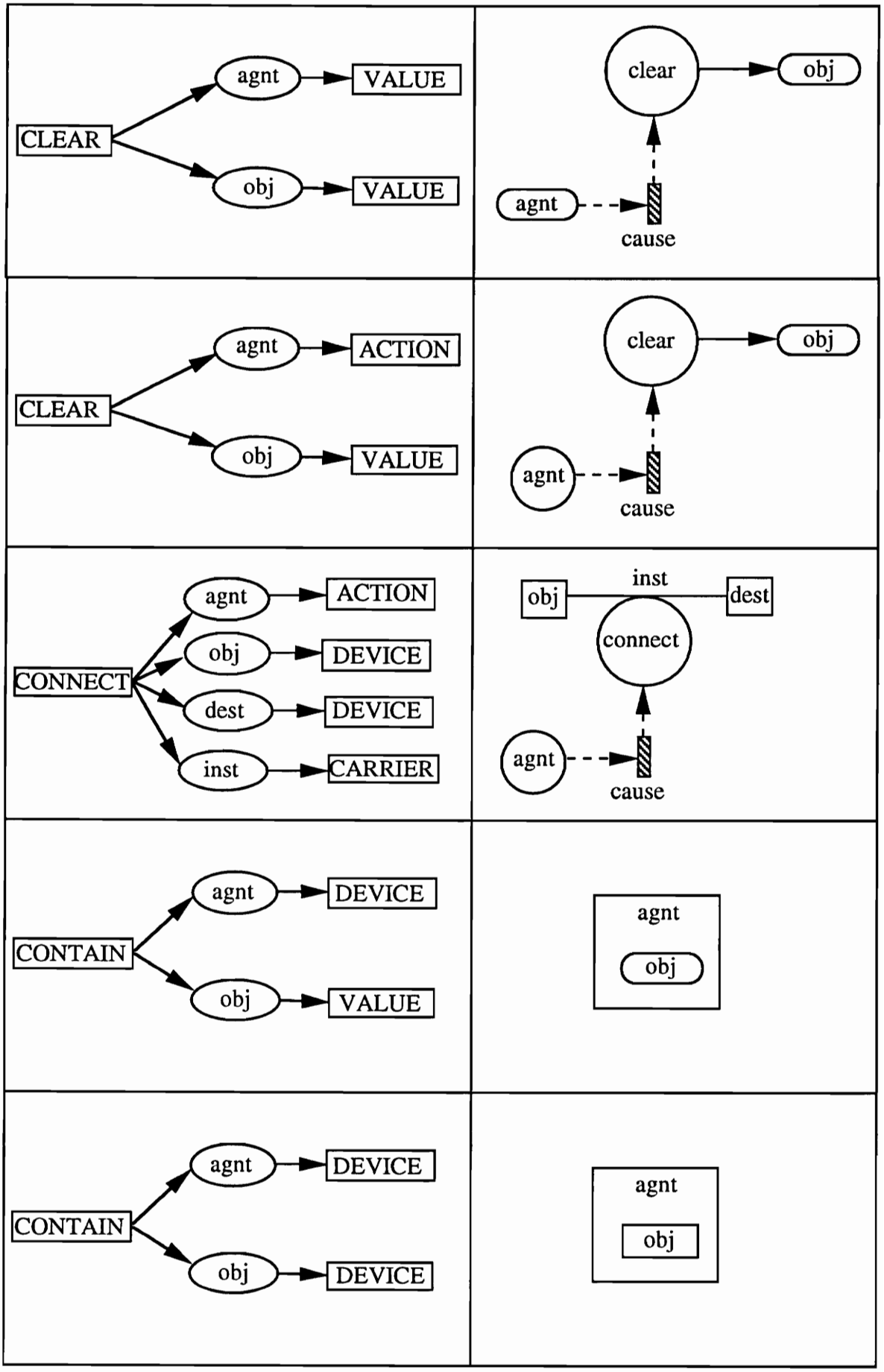
Appendix C. Interpretation Library

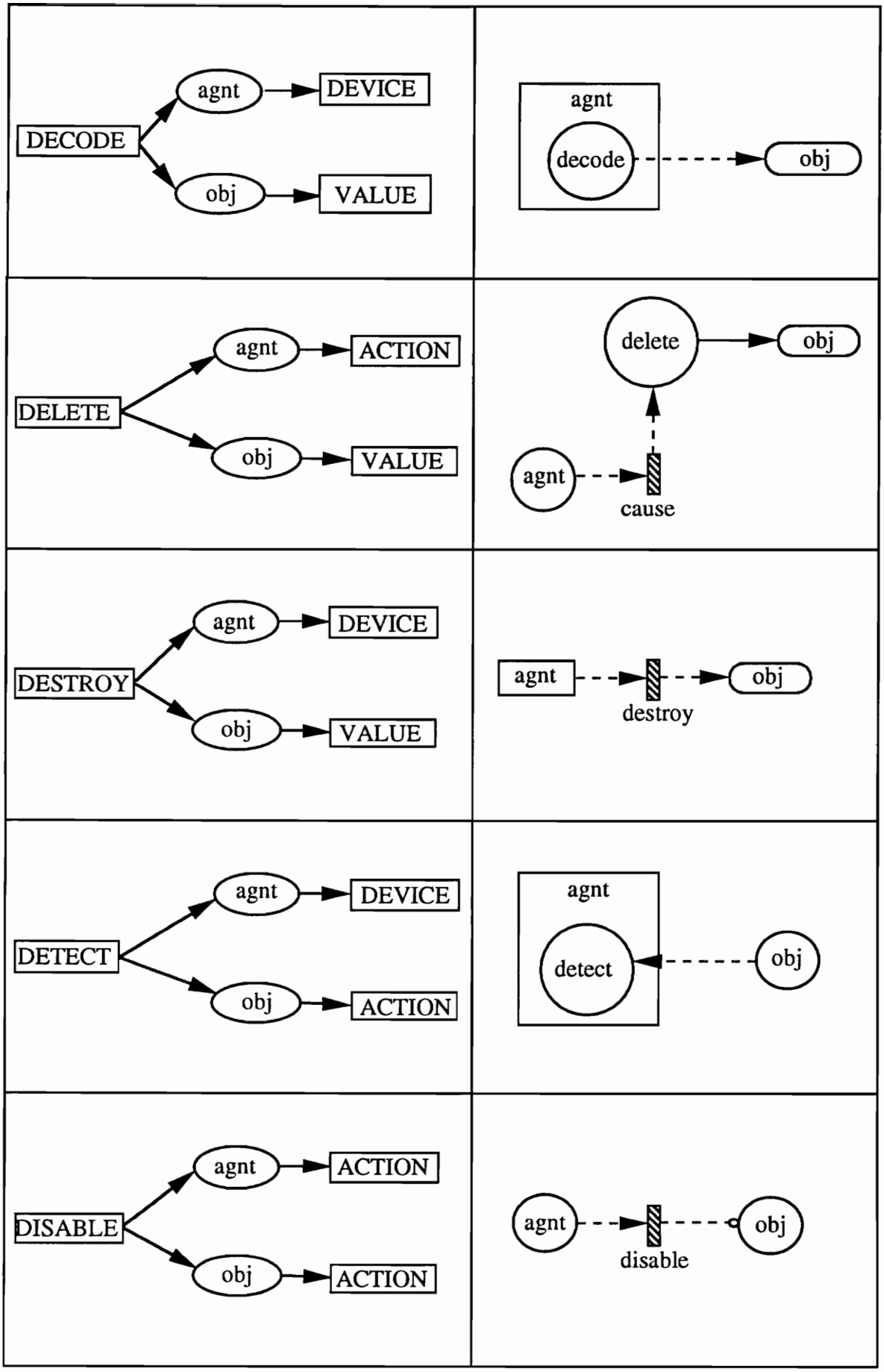
The canonical graphs and their canonical pictures in the Interpretation Library are presented below. The figures on the left are conceptual graphs and canonical pictures are on the right.

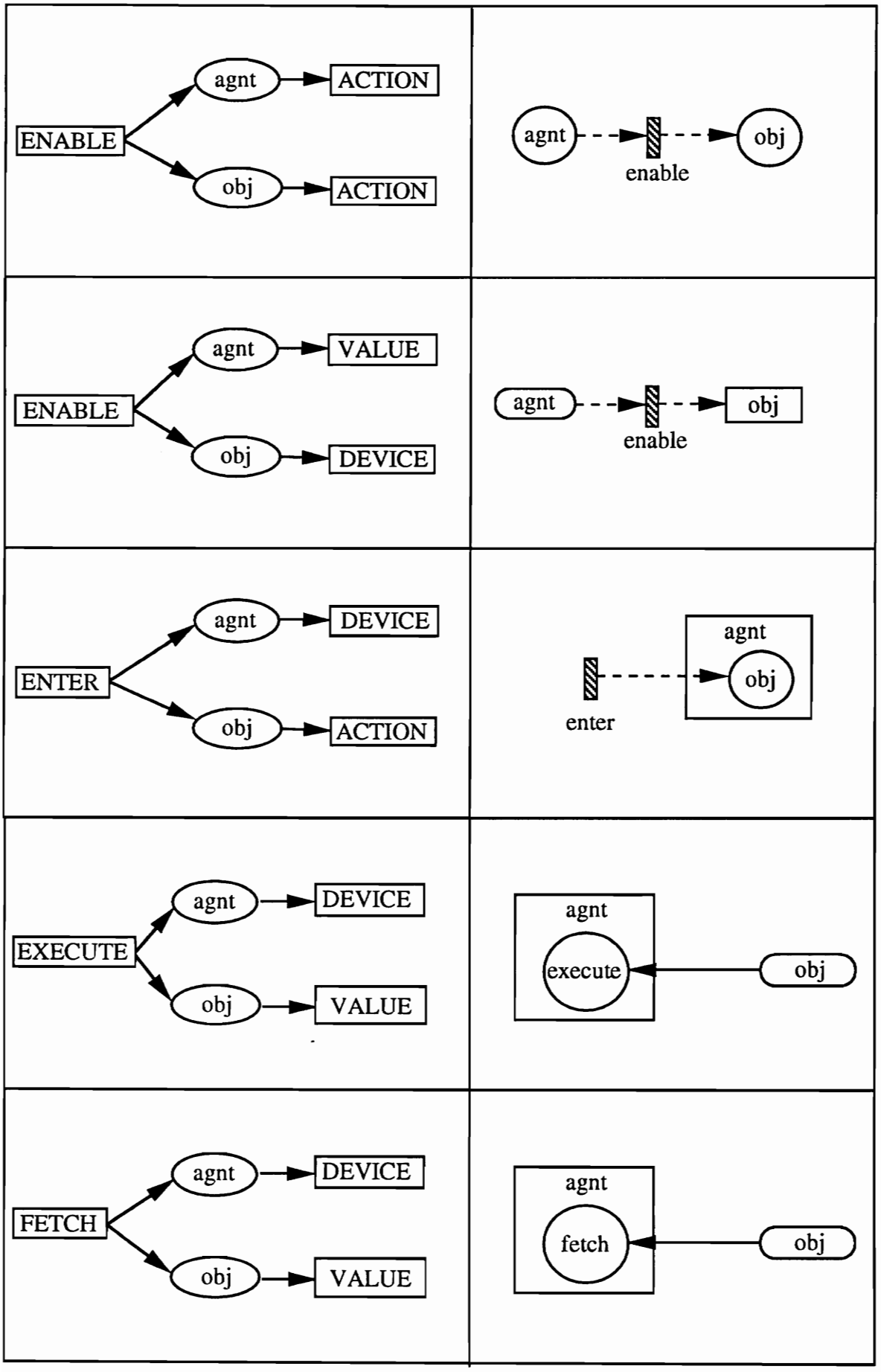

ACCEPT
agnt
DEVICE
obj
VALUE
agnt
accept
obj
ACCEPT
agnt
CARRIER
obj
VALUE
agnt
accept
obj
ACCESS
agnt
PROCESSOR
obj
VALUE
agnt
access
obj
ACCESS
agnt
PROCESSOR
obj
MEMORY
agnt
access
obj
ACCOMPLISH
agnt
DEVICE
obj
ACTION
agnt
accomplish
obj

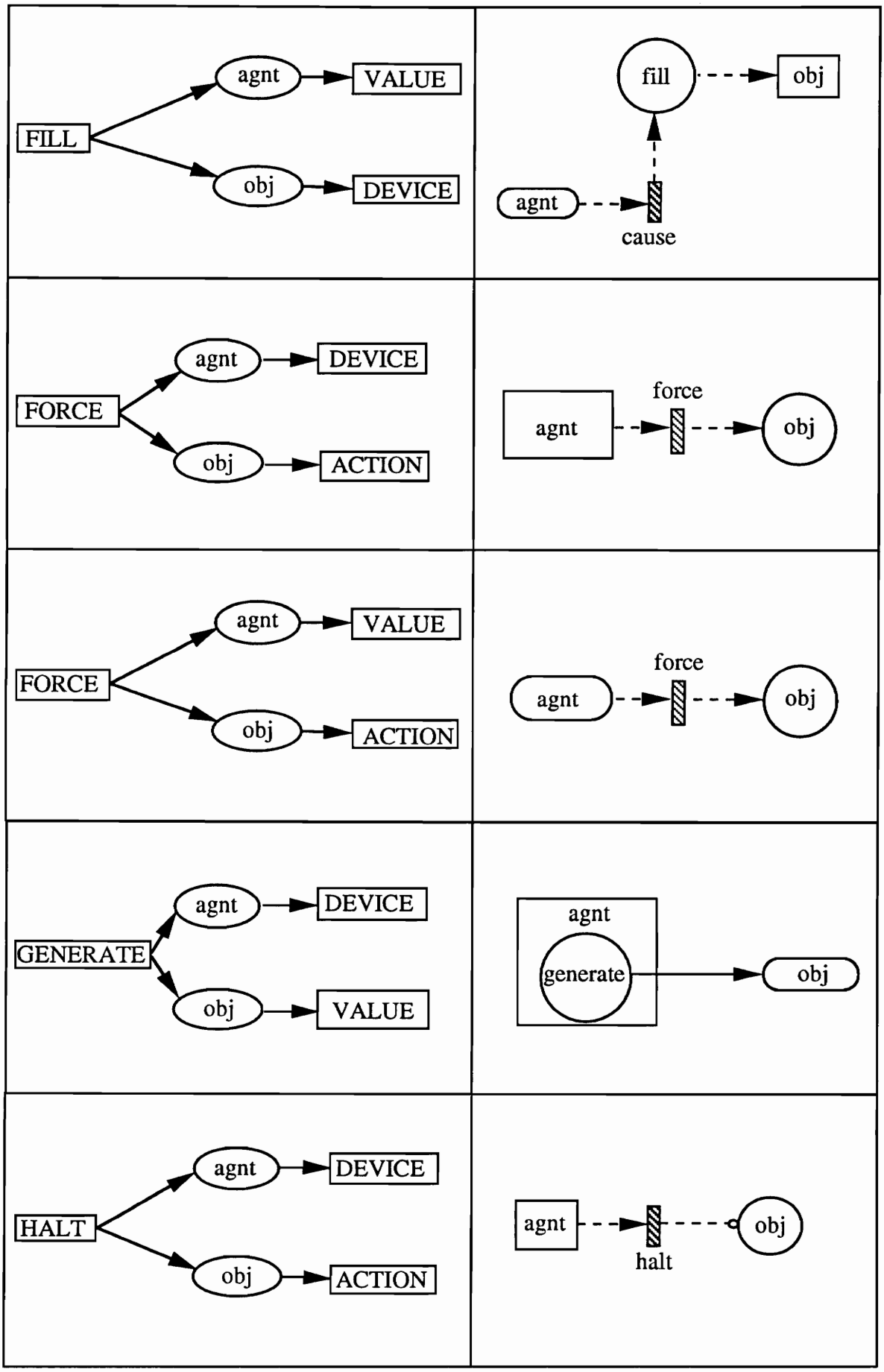

ACCOMPLISH
agnt
ACTION
obj
ACTION
accomplish
obj
agnt
cause
ACTIVATE
agnt
DEVICE
obj
DEVICE
agnt
activate
obj
ACTIVATE
agnt
VALUE
obj
DEVICE
activate
obj
agnt
cause
AFFECT
agnt
VALUE
obj
VALUE
agnt
affect
obj
AFFECT
agnt
VALUE
obj
DEVICE
agnt
affect
obj

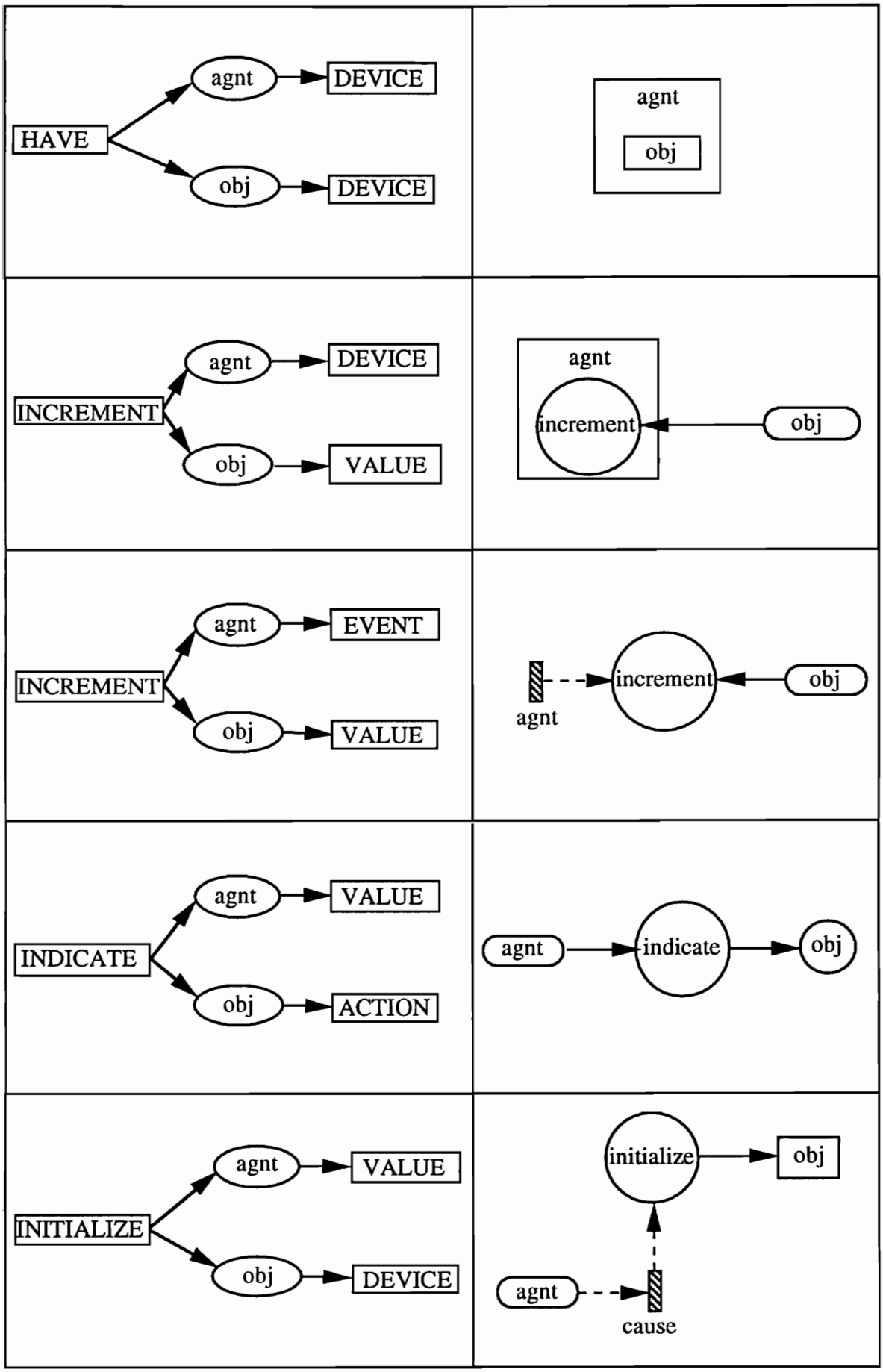

APPLY
agnt
obj
dest
DEVICE
VALUE
DEVICE
agnt
apply
obj
dest
APPLY
agnt
obj
dest
DEVICE
VALUE
CARRIER
agnt
apply
obj
dest
BEGIN
agnt
obj
DEVICE
ACTION
agnt
obj
begin
BRANCH
obj
dest
VALUE
VALUE
obj
dest
branch
CAUSE
agnt
obj
VALUE
ACTION
agnt
obj
cause

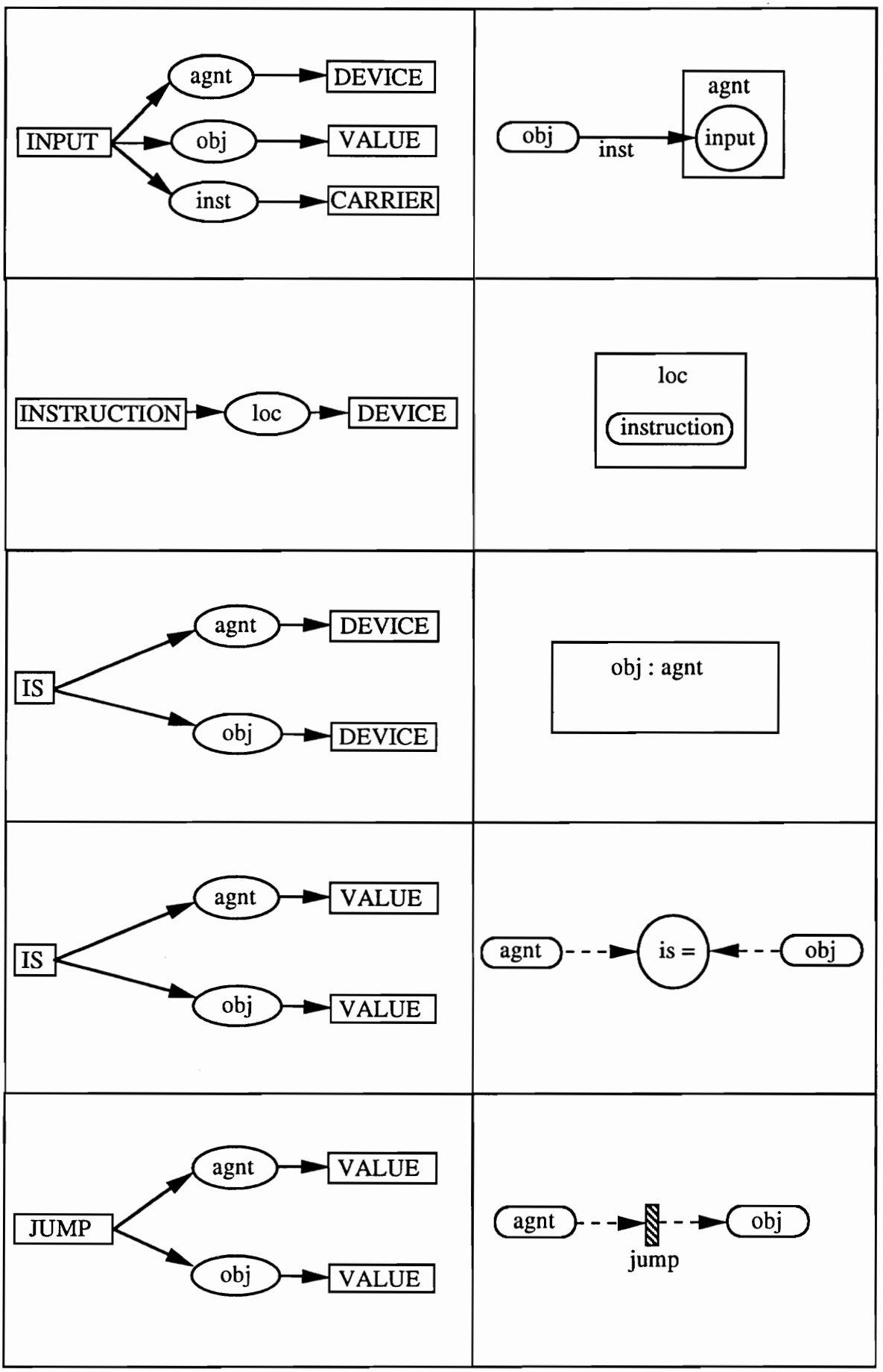

CLEAR
agnt
VALUE
obj
VALUE
clear
obj
agnt
cause
CLEAR
agnt
ACTION
obj
VALUE
clear
obj
agnt
cause
CONNECT
agnt
ACTION
obj
DEVICE
dest
DEVICE
inst
CARRIER
obj
inst
dest
connect
agnt
cause
CONTAIN
agnt
DEVICE
obj
VALUE
agnt
obj
CONTAIN
agnt
DEVICE
obj
DEVICE
agnt
obj

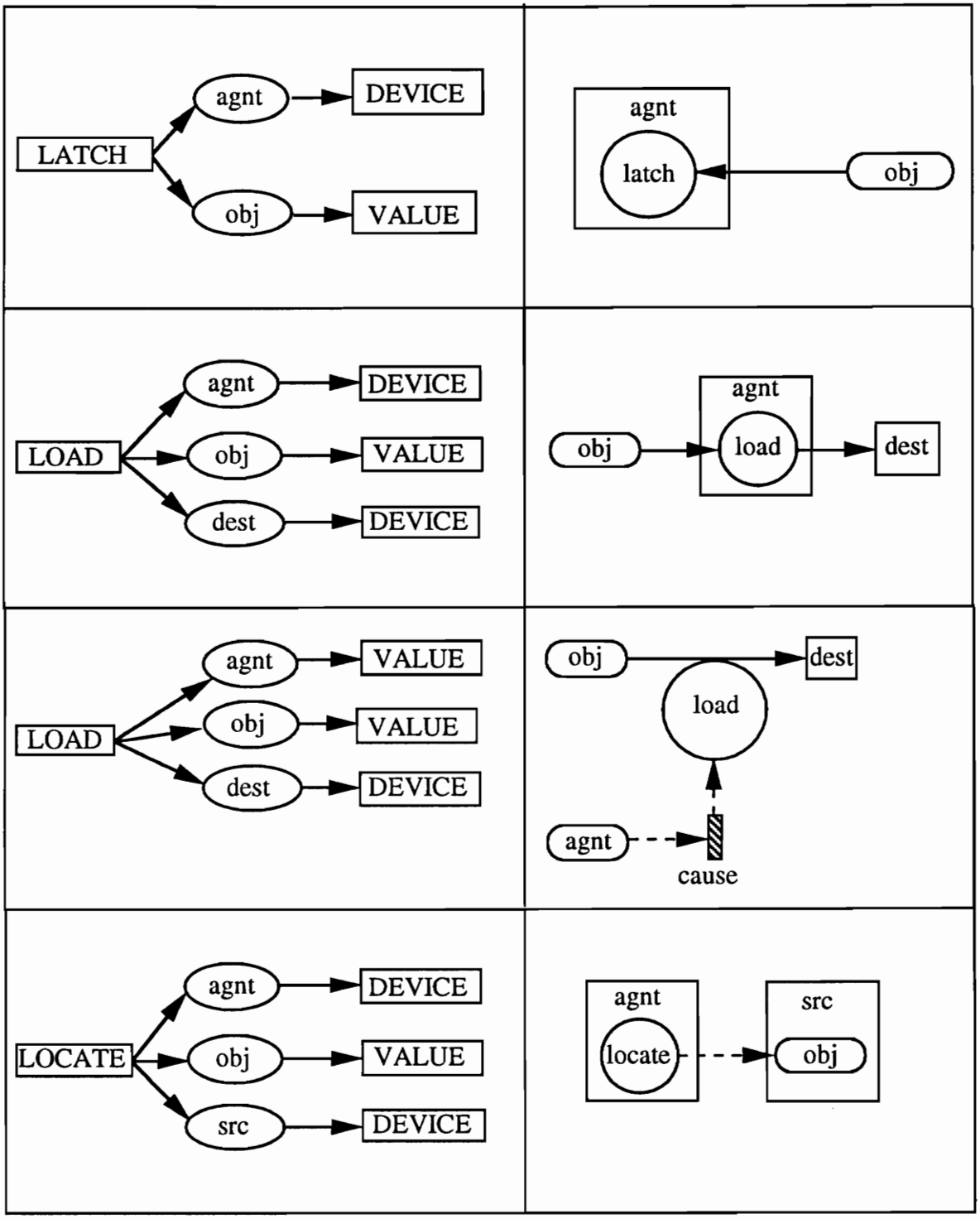

DECODE
agnt
DEVICE
obj
VALUE
agnt
decode
obj
DELETE
agnt
ACTION
obj
VALUE
delete
obj
agnt
cause
DESTROY
agnt
DEVICE
obj
VALUE
agnt
destroy
obj
DETECT
agnt
DEVICE
obj
ACTION
agnt
detect
obj
DISABLE
agnt
ACTION
obj
ACTION
agnt
disable
obj

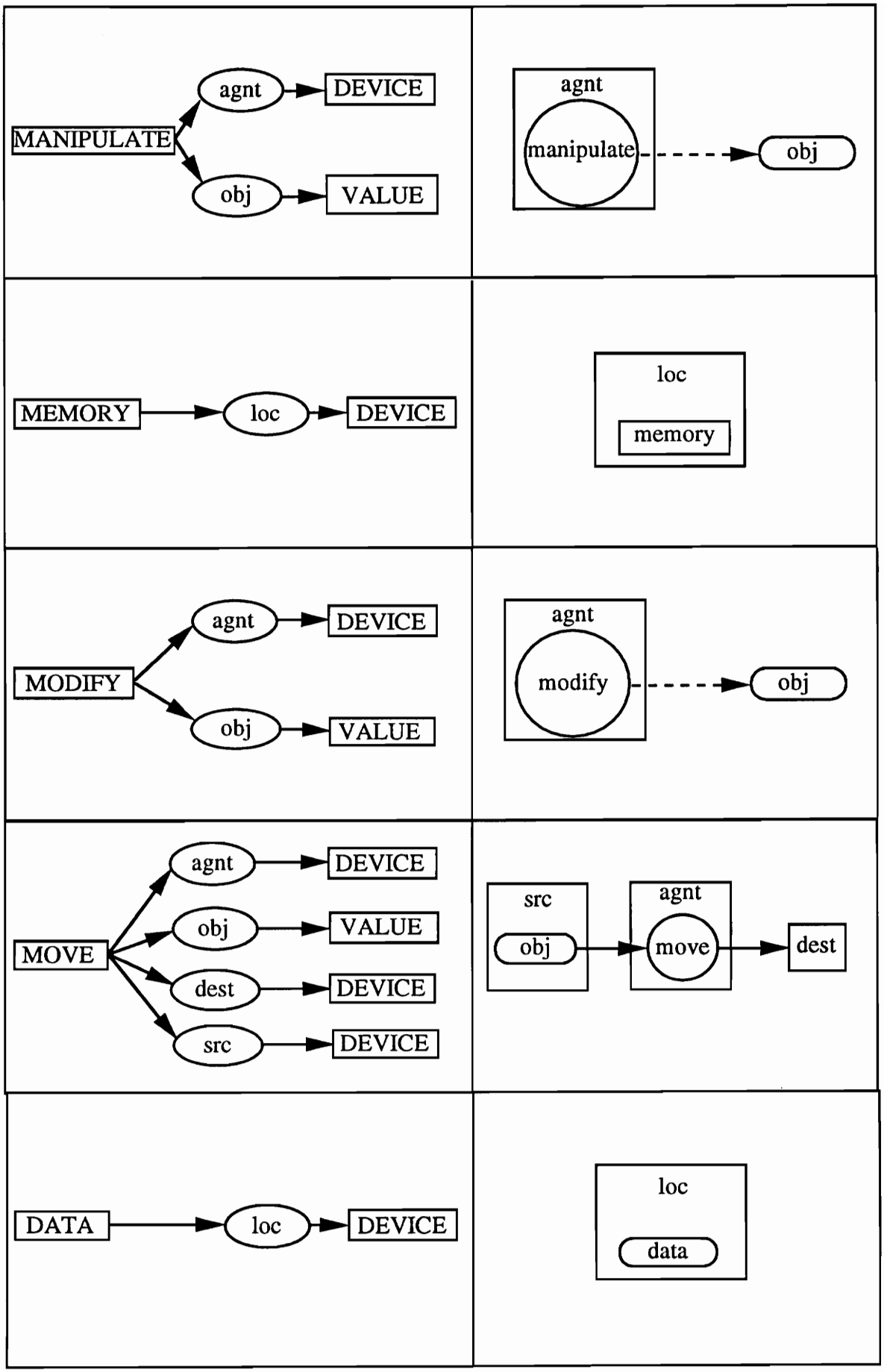

ENABLE
agnt
ACTION
obj
ACTION
agnt
enable
obj
ENABLE
agnt
VALUE
obj
DEVICE
agnt
enable
obj
ENTER
agnt
DEVICE
obj
ACTION
enter
agnt
obj
EXECUTE
agnt
DEVICE
obj
VALUE
agnt
execute
obj
FETCH
agnt
DEVICE
obj
VALUE
agnt
fetch
obj

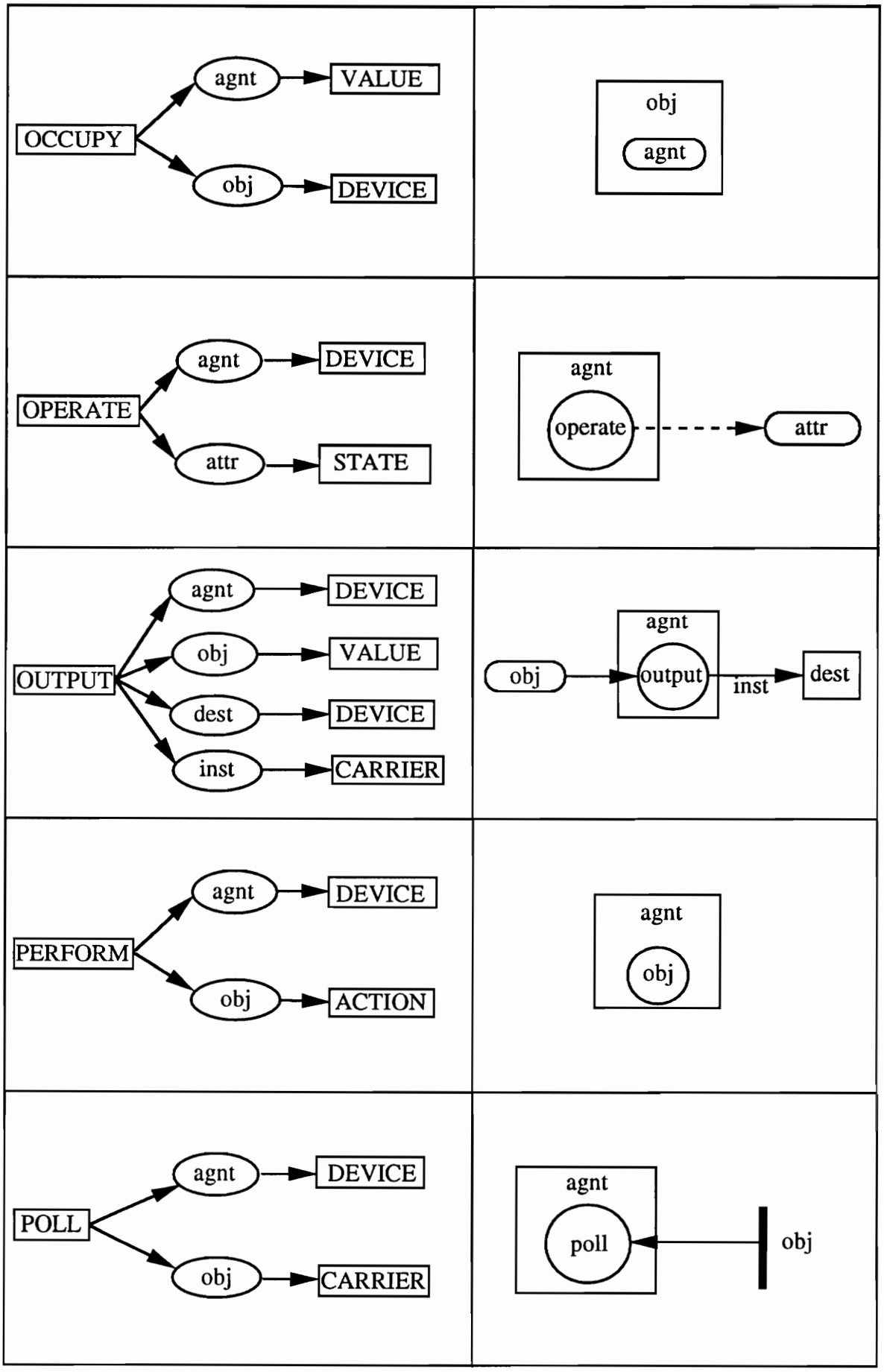

FILL
agnt
VALUE
obj
DEVICE
fill
obj
agnt
cause
FORCE
agnt
DEVICE
obj
ACTION
agnt
force
obj
FORCE
agnt
VALUE
obj
ACTION
agnt
force
obj
GENERATE
agnt
DEVICE
obj
VALUE
agnt
generate
obj
HALT
agnt
DEVICE
obj
ACTION
agnt
halt
obj

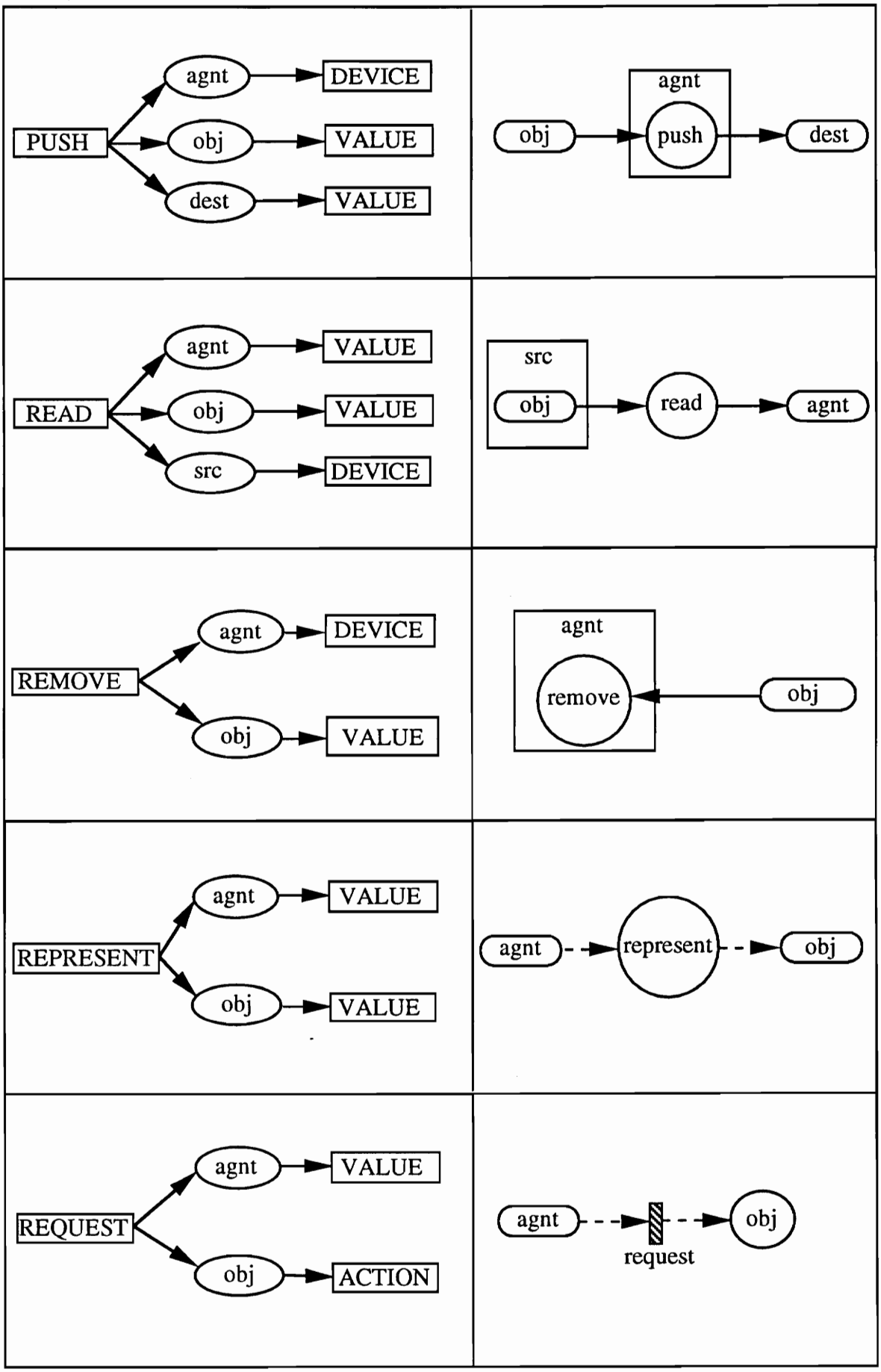

HAVE
agnt
DEVICE
obj
DEVICE
agnt
obj
INCREMENT
agnt
DEVICE
obj
VALUE
agnt
increment
obj
INCREMENT
agnt
EVENT
obj
VALUE
agnt
increment
obj
INDICATE
agnt
VALUE
obj
ACTION
agnt
indicate
obj
INITIALIZE
agnt
VALUE
obj
DEVICE
initialize
obj
agnt
cause

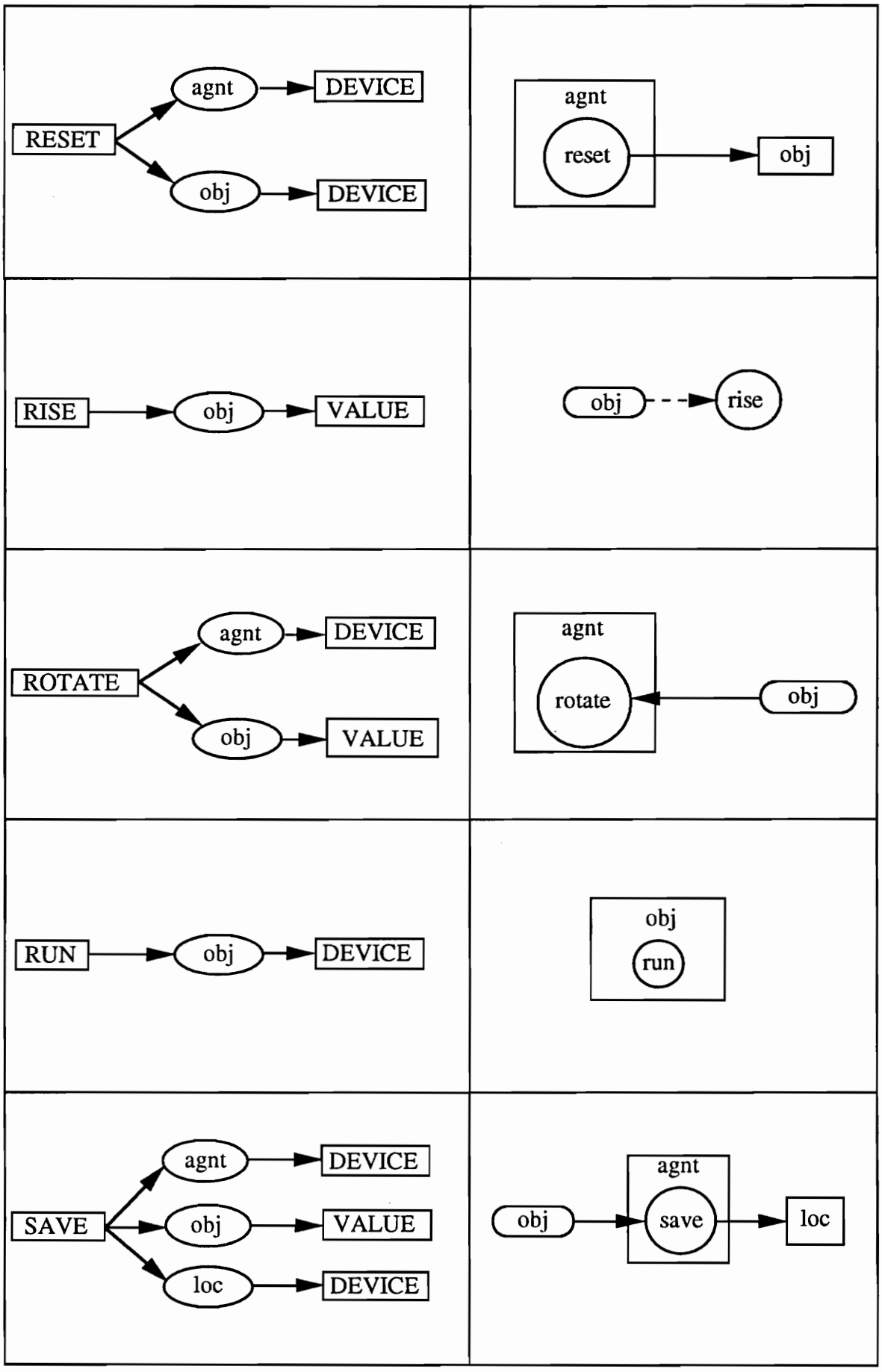

INPUT
agnt
obj
inst
DEVICE
VALUE
CARRIER
agnt
obj
inst
input
INSTRUCTION
loc
DEVICE
loc
instruction
IS
agnt
obj
DEVICE
DEVICE
obj : agnt
IS
agnt
obj
VALUE
VALUE
agnt
is =
obj
JUMP
agnt
obj
VALUE
VALUE
agnt
obj
jump

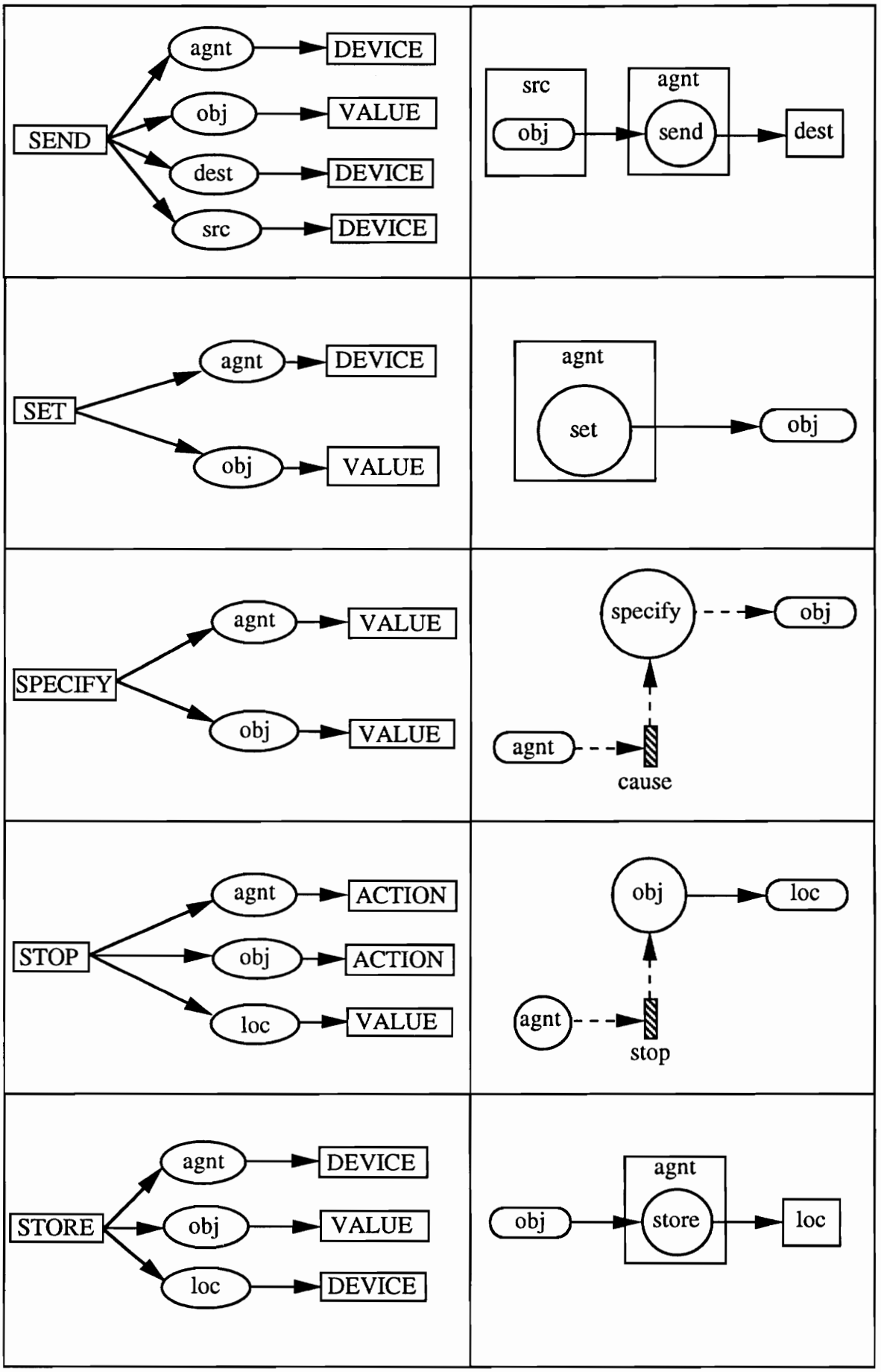

LATCH
agnt
obj
DEVICE
VALUE
agnt
latch
obj
LOAD
agnt
obj
dest
DEVICE
VALUE
DEVICE
agnt
load
obj
dest
LOAD
agnt
obj
dest
VALUE
VALUE
DEVICE
obj
load
dest
agnt
cause
LOCATE
agnt
obj
src
DEVICE
VALUE
DEVICE
agnt
locate
src
obj

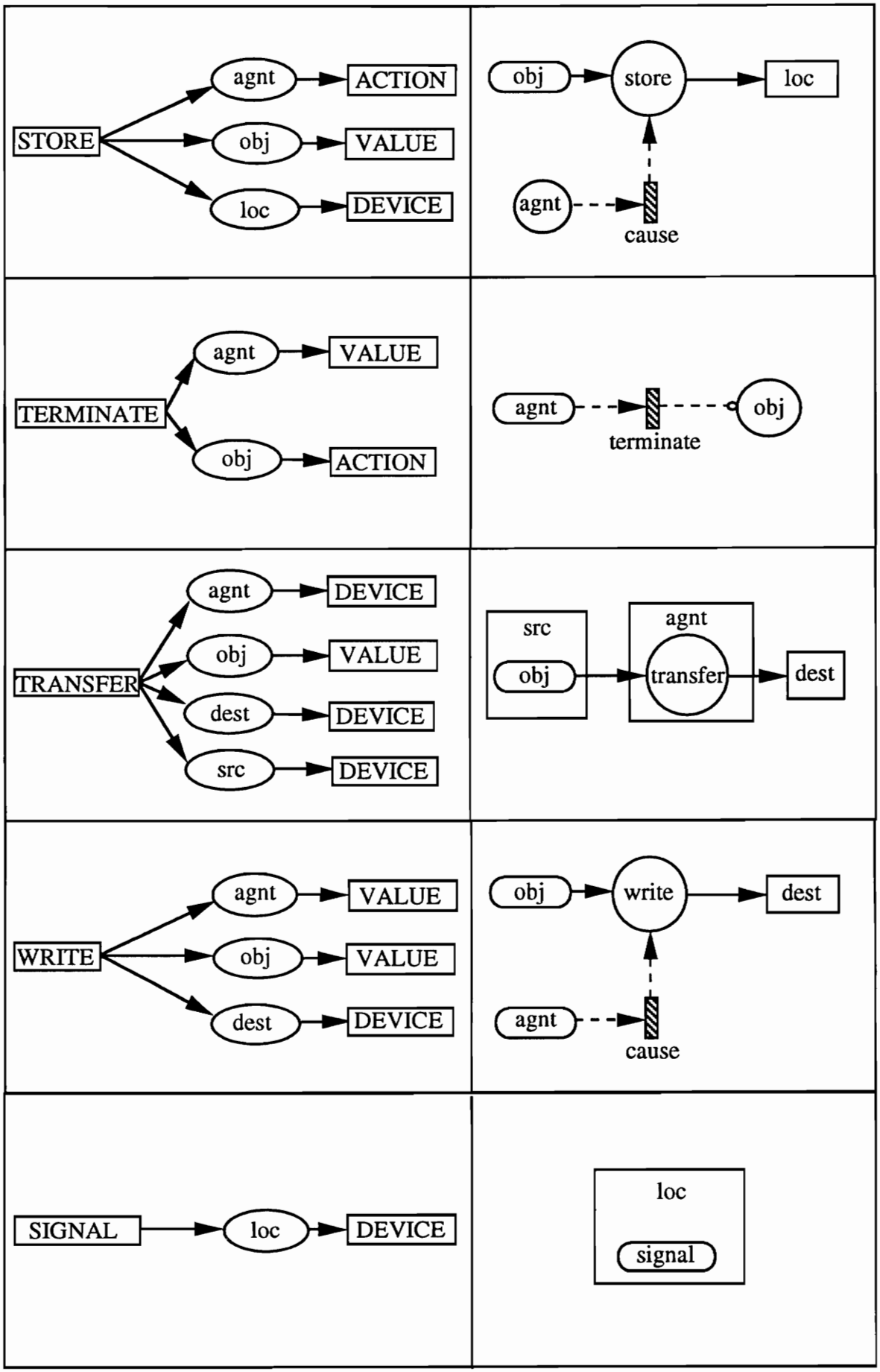

MANIPULATE
agnt
DEVICE
obj
VALUE
agnt
manipulate
obj
MEMORY
loc
DEVICE
loc
memory
MODIFY
agnt
DEVICE
obj
VALUE
agnt
modify
obj
MOVE
agnt
DEVICE
obj
VALUE
dest
DEVICE
src
DEVICE
src
obj
agnt
move
dest
DATA
loc
DEVICE
loc
data


OCCUPY
agnt
VALUE
obj
DEVICE
obj
agnt
OPERATE
agnt
DEVICE
attr
STATE
agnt
operate
attr
OUTPUT
agnt
DEVICE
obj
VALUE
dest
DEVICE
inst
CARRIER
obj
agnt
output
inst
dest
PERFORM
agnt
DEVICE
obj
ACTION
agnt
obj
POLL
agnt
DEVICE
obj
CARRIER
agnt
poll
obj


PUSH
agnt
obj
dest
DEVICE
VALUE
VALUE
agnt
obj
push
dest
READ
agnt
obj
src
VALUE
VALUE
DEVICE
src
obj
read
agnt
REMOVE
agnt
obj
DEVICE
VALUE
agnt
remove
obj
REPRESENT
agnt
obj
VALUE
VALUE
agnt
represent
obj
REQUEST
agnt
obj
VALUE
ACTION
agnt
request
obj


RESET
agnt
DEVICE
obj
DEVICE
agnt
reset
obj
RISE
obj
VALUE
obj
rise
ROTATE
agnt
DEVICE
obj
VALUE
agnt
rotate
obj
RUN
obj
DEVICE
obj
run
SAVE
agnt
DEVICE
obj
VALUE
loc
DEVICE
obj
agnt
save
loc


SEND
agnt
DEVICE
obj
VALUE
dest
DEVICE
src
DEVICE
src
obj
agnt
send
dest
SET
agnt
DEVICE
obj
VALUE
agnt
set
obj
SPECIFY
agnt
VALUE
obj
VALUE
specify
obj
agnt
cause
STOP
agnt
ACTION
obj
ACTION
loc
VALUE
obj
loc
agnt
stop
STORE
agnt
DEVICE
obj
VALUE
loc
DEVICE
agnt
obj
store
loc


STORE
agnt
obj
loc
ACTION
VALUE
DEVICE
obj
store
loc
agnt
cause
TERMINATE
agnt
obj
VALUE
ACTION
agnt
terminate
obj
TRANSFER
agnt
obj
dest
src
DEVICE
VALUE
DEVICE
DEVICE
src
obj
agnt
transfer
dest
WRITE
agnt
obj
dest
VALUE
VALUE
DEVICE
obj
write
dest
agnt
cause
SIGNAL
loc
DEVICE
loc
signal

Appendix D. Suite of Test Sentences

Presented in the appendix are sentences in the test suite for the Model Generator. These are taken from the file mgsuite.log that also contains the conceptual graphs for each sentence.

1. A memory write stores the data.
2. The chip has an on-chip, 8k cache.
3. Restting stops the execution of the instruction in memory.
4. The data to be stored in memory is incremented.
5. The processor is not a 16-bit device.
6. Resetting and initializing clear the memory.
7. Resetting, initializing and loading clear the memory.
8. The i80486 contains a cache, processor and co-processor.
9. All the internal registers are cleared.
10. The first 2 low input signals cause initialization.
11. The first 8 bytes of data are stored in memory.
12. All registers are cleared when the system resets.
13. The 2 low input signals initialize asynchronously if STRB is low.

14. The 8 bytes are stored asynchronously if STRB is low when the data arrives.
15. The machine can perform a read or write asynchronously if the RDWRITE signal is high.
16. The processor accepts the 8-bit data.
17. The processor accesses the memory M1 when chip-select is high.
18. The command activates the processor.
19. A long program affects the processor.
20. Application of a low value by the processor to the device clears it.
21. The processor begins execution of the program.
22. The program branches to the subroutine.
23. The EXEC command causes the execution of the program.
24. Execution of the OUT instruction connects the peripheral device to the processor on the 16-bit bus.
25. The memory contains the 16-bit address.
26. The decoder decodes the 16-bit address.
27. Initializing the memory deletes all the data on it.
28. The processor detects the generation of the faulty data.
29. The STRTCNT instruction enables the counter.
30. The processor fetches the data from the main memory.
31. The processor activates the counter which generates the 16-bit data.
32. The math co-processor halts the generation of faulty data.
33. Rise of the INC signal increments the data in the register.
34. The counter increments the 8-bit data if the INC signal is high.
35. A high RESET signal initializes the processor.
36. The processor inputs the data on the bus and generates a new instruction.
37. The processor latches the value if the STRB signal is high.

38. The processor loads the 8-bit data in the R1 register from the peripheral device.
39. A high LD signal causes the 8-bit data in the R1 register to be loaded into the peripheral device.
40. The processor locates the STARTUP program in the auxillary disk.
41. The cpu manipulates the data in the register.
42. The cpu moves the 8-bit data from register REG1 to register REG2.
43. The disk is occupied by the data.
44. The processor outputs the data to the IO port on the 16-bit bus.
45. The processor outputs the data to the IO port on the 16-bit bus if OUT signal is high.
46. The processor outputs the data to the IO port on the 16-bit bus if OUT signal is high when CLK rises.
48. The processor rotates the data in the register.
49. The rotated data is stored in the memory by the processor.
50. The DMA controller sends a block of data to the co-processor if the BG signal is high.

Vita

Aniruddha Thakar was born in Bhopal, India, on the 14th of July, 1966. He recieved a Bachelor of Engineering degree in Electronics Engineering from the University of Nagpur at Nagpur, India, in 1988. After that he began employment as a Research and Development Engineer with National Information Technologies Ltd., Bhopal, India.

Aniruddha decided to pursue a Master's degree in Electrical Engineering in 1991. He obtained the M.S. degree from Virginia Polytechnic and State University, Blacksburg, Virginia. At Virginia Tech he concentrated his studies in digital and VLSI design areas of Computer Engineering. He began employment at Intel Corporation, Chandler, Arizona, in August 1993.