BACK ANNOTATION FOR CONCEPTUAL STRUCTURES

by
Shreerekha Balachandar

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements of
Master of Science
in
Electrical Engineering

APPROVED

[Signature]

Dr. W. R. Cyre, Chairman

[Signature]

Dr. F. G. Gray

[Signature]

Dr. J. R. Armstrong

July 1995
Blacksburg, Virginia
BACK ANNOTATION FOR CONCEPTUAL STRUCTURES

by

Shreerekha Balachandar

Dr. W. R. Cyre, Chairman

Department of Electrical Engineering

(ABSTRACT)

The design of digital systems is getting more complex with rapid improvements in VLSI design which can accommodate many millions of gates in one integrated chip (IC). Additionally, the speed with which the design is completed is also becoming significant due to the demands of the market for the ICs. Tools to automate the initial design process can make the designer's task simpler and more accurate. The ASPIN system being built at Virginia Polytechnic Institute and State University focuses on deriving a synthesizable model for a digital system from various kinds of informal specifications (e.g. natural language descriptions, flowcharts, block diagrams, timing diagrams).

This thesis describes an interactive tool for validating and correcting formal models acquired from natural language specifications of digital system. Validation is important since the formal models have to be devoid of any ambiguities which might be present in the natural language specifications. The information acquired from the specifications is stored in an intermediate graphical notation called conceptual graphs. A preliminary tool called the Model Generator can produce a graphical display from conceptual graphs which helps the user visualize the model contained in the conceptual graph. The Back Annotator which is described in this thesis lets the user correct any misinterpretations by making changes to the graphical display such as additions, deletions, modifications, and movement
of objects in the display. Edits of the display are back annotated into the conceptual graph allowing the user to correct the interpretation.
ACKNOWLEDGMENTS

I express my deep gratitude to the Chairman of my committee and advisor Dr. Walling Cyre. His guidance and suggestions have been of immense help to me during the course of my thesis work. Apart from his assistance in my research work, he has also helped me in working towards the completion of my thesis with his timely replies on corrections and efforts to coordinate arrangements at Blacksburg. It has been a great privilege working under him.

I would like to thank Dr. J. R. Armstrong and Dr. F. G. Gray for serving in my graduate committee. Their cooperation played a big role in my final exam arrangements when I was communicating with them from across the country.

A special mention of Ms. Loretta Estes, Graduate Counselor needs to be put in this section for helping me in several ways during the course of my Master's degree at Virginia Tech.

My parents and brothers who have absolute confidence in my abilities have provided me with constant encouragement and support from India across the several thousands of miles that separate us. This work is dedicated to their intangible contribution towards my degree.

Finally, a big hearty thanks to my closest circle of friends whose advice and support through the past few months proved to be a great morale booster for me.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTERS

1. Introduction .................................................................................................................. 1
   1.1 Motivation .............................................................................................................. 1
   1.2 ASPIN system ....................................................................................................... 2
   1.3 Contributions to the ASPIN system .................................................................... 4
   1.4 Thesis Organization ............................................................................................ 4

2. Related Research ......................................................................................................... 5
   2.1 Visual Programming ............................................................................................ 5
   2.2 Visualization of Software Requirements .......................................................... 7
   2.3 Visualization scheme for a Medical Tutoring System ........................................ 9
   2.4 GRIT - A conceptual graph editor ..................................................................... 9
      2.4.1 Design of the GRIT ..................................................................................... 9
      2.4.2 Functionality of the GRIT ..........................................................................10
   2.5 Multiple Viewed Software Requirements with Conceptual Graphs ..................12

3. Background ............................................................................................................... 15
   3.1 Conceptual Graphs ............................................................................................... 15
      3.1.1 Concepts and relations ................................................................................ 16
      3.1.2 Concept type hierarchy ............................................................................... 17
      3.1.3 Canonical graphs ....................................................................................... 18
      3.1.4 Canonical rules ......................................................................................... 19
3.2 Semantics of Digital System .................................................. 23
3.3 Display ................................................................................. 26

4. Model Generator .................................................................... 29
   4.1 Overview of the Model Generator ......................................... 29
   4.2 Data Structures and Notations .............................................. 31
   4.3 Canonical Scripts .................................................................. 33
   4.4 Interpretation Library ........................................................... 34
   4.5 Model Generation Algorithm ............................................... 37
      4.5.1 The Mapping Algorithm ............................................... 37
      4.5.2 The Placer Algorithm .................................................. 39

5. Back Annotation ..................................................................... 41
   5.1 Overview of the Back Annotator ......................................... 41
   5.2 The Back Annotator ............................................................. 43
      5.2.1 Add Operation ............................................................. 43
      5.2.2 Delete Operation ........................................................ 49
      5.2.3 Modify Operation ....................................................... 52
      5.2.4 Move Operation ........................................................ 57
   5.3 Algorithm of the Back Annotator ........................................ 60
      5.3.1 Add Algorithm ........................................................... 60
      5.3.2 Delete Algorithm ....................................................... 62
      5.3.3 Modify Algorithm ...................................................... 64
      5.3.4 Move Algorithm ........................................................ 65
   5.4 Implementation of the Back Annotator .................................. 66

6. Back Annotation Examples .................................................... 72
   6.1 Add Utility Example .......................................................... 72
6.2 Delete Utility Example.................................................................77
6.3 Modify Utility Example..............................................................82
6.4 Move Utility Example.................................................................86

7. Future Work..................................................................................90
  7.1 Graphical Capture System.........................................................90
  7.2 Back Annotation and Model Generation for Physical Systems........90
  7.3 Animation in Display.................................................................91

REFERENCES

APPENDIX - User's Manual
LIST OF FIGURES

1.1 Overview of the ASPIN System Objective.................................................2
2.1 Process Model Graph in Modeler’s Assistant........................................6
3.1 Pictorial Conceptual Graph.................................................................16
3.2 Textual Conceptual Graph.................................................................17
3.3 Type lattice in a conceptual system.....................................................18
3.4 Canonical graph for READ and DATA.................................................19
3.5 Canonical graph after restriction of value to DATA............................20
3.6 Two Canonical graphs to be merged...................................................20
3.7 Canonical graph of READ after join..................................................21
3.8 (a) Canonical picture (!) for INCREMENT...........................................22
3.8 (b) Canonical picture (2) for INCREMENT...........................................22
3.9 Partial Conceptual type hierarchy......................................................24
3.10 Object set for the digital display scheme..........................................27
3.11 Literal Display......................................................................................28
3.12 Schematic Display example...............................................................28
4.1 Overview of the Model Generator.......................................................29
4.2 (a) Canonical graph (1) for INCREMENT............................................34
4.2 (b) Canonical picture and script (1) for INCREMENT...........................34
4.3 (a) Canonical graph (2) for INCREMENT............................................35
4.3 (b) Canonical picture, script (2) for INCREMENT..................................35
4.4 Flowchart of the Mapping Algorithm................................................37
4.5 Flowchart of the Placer Algorithm.......................................................39
5.1 Multiple Interpretations for a single sentence......................................43
5.2 Display generated by Model Generator (before Add)..........................46
5.3 Display after Add (Step 1).....................................................................47
5.4 Display after Add (Step 2) .............................................................. 48
5.5 Display generated by Model Generator (before Delete) .................. 49
5.6 Display and Conceptual Graph after deletion .................................. 51
5.7 Display and Conceptual Graph before "Modify" ............................... 54
5.8 Display and Conceptual Graph after "Modify" .................................. 56
5.9 Display before "Move" ................................................................. 58
5.10 Display after "Move" ................................................................. 59
5.11 Block Diagram of the Back Annotator Implementation .................... 67
5.12 Icons and Connectives permitted in the Back Annotator .................. 69
5.13 Display Window of the Back Annotator ...................................... 70
5.14 ADD Menu ............................................................................ 71
5.15 DELETE Menu ........................................................................ 71
5.16 MODIFY Menu ....................................................................... 71
5.17 MOVE Menu ......................................................................... 71
6.1 Original Display generated by Model Generator for Add Example .... 74
6.2 Edited display after Add Operation .............................................. 75
6.3 Original Display generated by Model Generator for Delete Example .. 78
6.4 Edited display after Delete Operation ......................................... 79
6.5 Original Display generated by Model Generator for Modify Example .. 84
6.6 Edited Display after Modify Operation ......................................... 85
6.7 Original Display generated by Model Generator for Move Example .... 88
6.8 Edited Display after Move Operation ........................................... 89
LIST OF TABLES

1. Definition of Conceptual Types................................................................. 23
2. Conceptual relations and their definitions............................................... 25
3. Relations for possible connectives......................................................... 45
4. Relations for possible icon containments............................................. 46
Chapter 1. Introduction

1.1 Motivation

With the advancements in VLSI Design and IC manufacturing processes, the digital systems that are being designed are becoming larger and more complex. Designing such a large complex system involves a considerable amount of time and effort. To reduce the time for the design phase of a system and to make the designer's task simpler, an automated tool to help design a system would be very useful. There are tools in the market to synthesize a system from formal engineering models. To make automated synthesis tool more versatile, it would have to handle more common forms of specifications. Some of the frequently encountered forms are natural language specifications, block diagrams, timing diagrams, and flow charts. A tool which can take in these types of specifications as input and produce a model which can be used in simulation and synthesis would be very appropriate in the rapidly developing field of Computer Aided Design.

The work discussed in this report is a part of an automated design tool which aims to extract engineering models for synthesis from natural language specifications. Natural language specifications have ambiguities. When synthesis is done based on models developed automatically from natural language specifications, the designer has to be assured that any such existing ambiguities will be satisfactorily resolved. The motivation for the work reported here is to provide a tool for validating and correcting the automatic interpretation of natural language specification using graphic visualization and back annotation.
1.2 ASPIN SYSTEM

The Automatic Specification Interpreter (ASPIN) [2] is a system-level design automation system to develop formal engineering models from informal specifications used to define the requirements of a digital system. The engineering models which can be simulated or synthesized, are expressed in VHSIC Hardware Description Language (VHDL). The ASPIN system figure Fig 1.1 gives an overall view of the structure of the system. The first step in the interpretation process is to build a knowledge representation of the information in the system specifications. The knowledge representation formalism used in ASPIN system is a type of semantic net called conceptual graphs.

![Figure 1.1 Overview of the ASPIN system objective](image)
Informal specifications may be represented in natural language specifications, flow charts, timing diagrams, block diagrams. The present work is primarily concerned with natural language requirements. Natural language specifications in the form of English sentences are input to the Parser which generates parse trees reflecting the structure of the sentences. When a sentence is ambiguous, it may have multiple parse trees. These parse trees are used by the Semantic Analyzer to create conceptual graphs. Ambiguities in the meanings of words can result in a parse tree generating more than one CG. Only one CG is generated for a sentence even if the sentence is ambiguous. This conceptual graph is fed into the Model Generator to visualize the knowledge content of the conceptual graph on a display for validation by the user. If the interpretation shown in the graphical display is incorrect, the specification author makes corrections to the display. These corrections are back annotated into the conceptual graph by the Back Annotator. The display consists of icons and connectives. Once the graphical display is generated, the specifications author can check the correctness and edit the display by adding, deleting, moving or modifying the icons or connectives. These changes are back annotated into the conceptual graph by modifying the conceptual graph to correspond to the corrected graphical display. Once this feedback process is complete, the conceptual graph is integrated with the graphs from other sentences and passed through the Linker which generates VHDL models. VHDL models can be accepted by a tool called Modeler's Assistant for compilation and refinement by a designer. The VHDL models can also be used for simulation, analysis or synthesis.
1.3 CONTRIBUTIONS TO THE ASPIN SYSTEM

The contributions of this work to the ASPIN system are

- Revision of the Model Generator from an X-Lib based system to an X-View based system
- Implementation of the graphic editor for the graphical display system
- Implementation of back annotation for the graph validation process in the C programming language

1.4 THESIS ORGANIZATION

The first chapter gives an introduction and the motivation for the work reported. Chapter 2 gives a summary of other related research. The third chapter deals with the background knowledge about Conceptual graphs and the digital scheme of representation. The fourth chapter describes the Model Generator and the background needed to understand the generation of the display. Chapter 5 deals with the Back Annotator with all the inner modules, the algorithm, and the implementation of the Back Annotator. Examples illustrating back annotation and how the back annotation feature works are included in Chapter 6. The final chapter contains suggestions to improve the Back Annotator to accommodate more modifications to the conceptual graph depending on an evolving ontology and to use Back Annotator and the Model Generator as a more general tool for systems other than digital systems. Depicting system behavior or simulation using animation also is discussed. The Appendix contains an user manual for the Model Generator and the Back Annotator.
Chapter 2. Related Research

In this chapter, some published research ideas which are related similar to back annotation and visualization using graphic display is discussed. Visualization is defined as the viewing of numerical or complex information in a pictorial form to make analysis easier. An increasing number of projects are being developed to include visualization and graphical user interfaces rather than relying only on text. Visual programming is gaining more ground because of the inherent human ability to absorb visual information quicker.

2.1 Visual Programming

This section describes a few systems which use visualization and are built on the concept of graphical user interfaces. The Modeler's Assistant [12] is a graphical user interface to develop behavioral VHIC Hardware Description language (VHDL) models for digital systems. The Modeler's Assistant, similar to Back Annotator, is a graphical user interface based design tool used to expedite the design cycle of digital systems. The graphical interface of the Modeler's Assistant is called Process Model Graph (PMG). A PMG consists of nodes and arcs drawn between the nodes. The nodes signify processes and the arcs denote signals. A process in VHDL, graphically represented by a circle in the PMG, is defined as a set of statements describing the behavior of a module in the system. Input and output signals are denoted by smaller circles on the periphery of the process circle. Figure 2.1 shows an example process model graph on Modeler's Assistant interface. When a system is modeled, a library of primitive models is referred to. Primitives can be chosen from the existing library or new primitives can be defined and included in the
library. The module to be designed can be graphically input by the user and Modeler's Assistant is capable of generating the VHDL code for this input. The user can name the ports of a process and the names in the primitives are substituted with the user specified names. Using the primitive models in the library, any behavioral description can be generated from the graphical display.

![Diagram of the Modeler's Assistant interface]

**Fig 2.1** Process Model graph in Modeler's Assistant
2.2 Visualization of Software System Requirements

Visualization is also used for specifying software system requirements. Consens [2] describes a query language for visualizing and modifying software system requirements in the design phase of a software. Visualization is applied to analyze problems involving large sets of objects and complex relationships between them in software design. The query language named GraphLog can serve to visualize inter-component dependencies and code overlay structure in the memory. The entire software system is modeled as a directed graph. A package in the software is a group of modules put together under a single name. The icons in the directed graph signify packages and the arcs between them show the dependency of one package over the other. The dependencies are calls from one package to the other. A query about the system is drawn as a graph and the tool matches the input graph with like patterned graphs describing the requirements in the database. The query might further lead to retrieval of particular instances from the matched graphs. The dependencies between packages may give rise to cycles in the graph. To improve the design, methods can be suggested to reduce the number of cycles by modifying the structure of a package thereby decreasing dependencies. The user can input changes to the design by graphically modifying the icons on the display. This changes the underlying representation of the system requirements. The Back Annotator shares this way of implementing changes in the original specifications through a graphical interface. The digital system is represented as a graphical display and the user makes changes to this display which are automatically back annotated into the internal form of the system specifications.
2.3 Visualization scheme for a Medical Tutoring System

Another visualization scheme by Sukthankar [10] for a medical tutoring system (CIRCSIM_TUTOR_V2) uses a graphical user interface to visualize information about the effects of perturbations to the cardio-vascular system. This example illustrates the wide spectrum of domains that use visualization for interpreting knowledge bases. The display comprises of icons and connectives. The icons are semantically associated with physiological variables and connectives show the dependencies of the variables over one another. A student using the tutor can build a display called a Concept Map. The system verifies the knowledge contained in the student's concept map. A menu of icons is displayed for the student to choose from while drawing the concept map. The Back Annotator in this thesis lets the user build a display and generates the conceptual graph represented by the display. The user has a set of icons to choose from while constructing the display but these icons are not always displayed. Options which let the user specify the type of display element to be added are provided.

The important difference that needs to be pointed out between the Back Annotator and the Modeler's Assistant type of tools and the latter two visualization schemes that are described in this section is that the former two do not have isomorphic (one-to-one) mappings between the display and the library or database. This makes visualization and back annotation more complicated than in the case of a one-to-one mapping wherein every object in the display directly corresponds to an entity in the database. Addition and deletion of display objects in the case of isomorphic mapping effects a similar operation on the entity corresponding to the object. In non-isomorphic mappings changes to the display may affect neighborhoods in the underlying knowledge base.
2.4 GRIT- A conceptual graph editor[8]

The Graph Implementation Tool (GrIT) is a graphical user interface designed for a conceptual graph project called PEIRCE. The PEIRCE group is involved in research to widen the applicability of conceptual graphs. This graphic interface provides detailed drawing, editing and browsing facilities for conceptual graphs in addition to various logical operations on them.

The PEIRCE project aims to implement a standardized, freely distributed portable graph environment. The main module of the PEIRCE software is a database from which knowledge can be retrieved through queries. Since graphical interfaces are a convenient way to handle large databases, the Graph Implementation Tool has been developed as a part of the PEIRCE project.

2.4.1 Design of the GrIT [8]

GrIT functions as a drawing tool for the construction of conceptual graphs and as an interface to the PEIRCE database. The tool can be divided into three modules. A user interface module handles the user interactions (mouse and keyboard inputs) on the graphic display. This interface is isolated from the main function module so as to make the implementation interface independent. The main function module is the core of the software which interacts with the database. The third module is the graphics interface which draws the conceptual graph pictorially. This module can also be redesigned depending on the graphics package that is chosen. The Back Annotator has a similar
organization with an X-View based User interface module, back annotation module and the X-Lib graphics module.

2.4.2 Functionality of the GrIT [8]

The GrIT provides drawing, viewing, editing, and database operations. Each of these will be briefly described in this section.

- The drawing operation facilitates the graphical representation of a conceptual graph on the display. The choice of adding a concept or relation is specified and the required text is entered. Following this, in the case of a concept, the node is positioned on the display with a mouse click. Connectives are added by choosing the source and destination icons.

- Viewing operations provide user with the facility to scroll and zoom view large graphs that are displayed.

- Editing allows both local and global manipulation of conceptual graphs. The user may reposition a node by a click and drag action with the mouse placed on the node which is to be moved. The position of all arcs attached to the node are recalculated and the arcs are redrawn whenever the node itself is redrawn. Deletion of arcs and nodes are also possible. Deletion of a node results in the automatic deletion of all arcs connected to the node. The GrIT has additional features referred to as Move Block and Delete Block
which permits move and delete operations on large portions (called blocks) of conceptual graphs.

- The GrIT is a graphical interface to the conceptual structures database. Once a well formed conceptual graph has been constructed, the linear form may be generated. This linear form can be added to the database. Also a query can be made in the graphical form and all linear forms which contain this graph as a substructure are displayed.

- The file operations in GrIT allow the user to save a conceptual graph entered graphically and later reload it. Also there is a partial load facility which allows an arbitrary number of graphs to be incrementally loaded into GrIT from files. This facility lets the user independently develop many individual graphs, store them and construct more complex graphs by combining suitable selections of the existing graphs using partial load.

- Concepts in conceptual graphs are ordered in a type lattice. A type lattice in its linear form specifies the supertype and subtype of every concept. The linear form is used to facilitate long term storage. Such a type lattice can be loaded into the GrIT and the subtypes and supertypes of any concept can be found by selecting the relevant option from the Type Lattice menu and clicking on the required concept. Maximum common subtype or minimum common supertype of two selected nodes can also be found out.

- Finally canonical operations restrict and expand are currently implemented in the GrIT. These operations are important in graph unification and graph matching because they provide a means of specializing (restrict) and generalizing (expand) the label associated with a concept node.
The Back Annotator provides similar operations on the display with an interface to the
database. Add operation corresponds to the drawing facility in the GrIT. Back Annotator
is capable of moving and deleting display elements which is the same as the editing
operations in this tool. The Modify operation in the Back Annotator is a form of the
"restrict" canonical operation.

2.5 Specifying Multiple Viewed Software Requirements with Conceptual Graphs

Delugach[7] discusses software requirements specification and transforming the
specifications in different formats into a common notation. The need for a common
notation stems from the fact that all requirements of a system cannot be expressed clearly
in a single notation because of the limitations in every notation. On the other hand, when
there are several notations, the specifications should be easy to analyze and compare with
each other. The solution to deal with such a situation is to translate different notations into
a common form which can be easily stored and analyzed for comparison. The common
notation chosen in this paper is conceptual graphs (termed R-spec graphs). The idea of
transformation from a few specific notations (chosen on the basis of their regular usage in
software system requirements specification) to R-spec graphs and back are given briefly.

- Entity relation diagrams (E-R diagrams) consists of entity and relation nodes which are
  connected by directed arcs. They are similar to conceptual graphs in structure and the
  mapping takes an entity to a concept and a relation in the E-R diagram to a relation in
  the R-spec graph. The direction of relations in the R-spec graph are determined with
the help of predefined canonical graphs. The mapping in this case is straight forward resulting from the structure of E-R diagrams and R-spec graphs.

- Data flow diagrams (DFDs) consists of bubbles representing processes connected by labeled arrows denoting the flow of data between the bubbles. In order to represent processes in DFDs, an actor node is defined in the R-spec graphs. These actor nodes are capable of changing the referents of their output concepts based on their input concepts. While transforming a DFD into an R-spec graph, a process becomes an actor node and a data flow arc from process A to B with label L becomes a conceptual graph concept of type L linked to actor B and from actor A. The actor node is very significant since processes cannot be adequately represented otherwise.

- State transition diagrams consists of state nodes connected by directed arcs labeled with the inputs that cause the state transition (termed input events) and the outputs resulting from the input and state combination (termed output events). To transform these state diagrams to R-spec graphs, State concepts are defined and special nodes called demons are introduced by the author. These demons signify state transition as elimination of the previous state and creation of a new state. The demon nodes which are labeled transition have input nodes connected to them to cause the transition. In cases where outputs are generated, output concept nodes are connected to the transition node.

- The fourth type of notation used for translation is the SREM R-net which is a semantic network used for representing software specifications. The translation is based on a component-to-component equivalence table which shows the R-net components and
the conceptual graph representation corresponding to it. This table is similar to the library that the Model Generator and the back Annotator uses to generate the display and modify the same. The R-Net has symbols for Start, message, process, connectors (an AND relation between processes) and selectors (an OR relation between processes) and end of the net. Messages are mapped to concepts and processes are mapped to actors. The connector and selector are mapped to conceptual graph contexts consisting of actors and processes. This mapping is not a purely isomorphic case. The translation is done by referring to the symbol mapping table for every component of the input specification and replacing it with the corresponding match.
Chapter 3. Background

The representation scheme used in this thesis work to interpret digital system specifications is conceptual graphs. This chapter deals with definitions and rules in conceptual graph theory. The relevant background on digital system specifications and display schemes are also discussed.

3.1 Conceptual Graphs

Conceptual graphs, a knowledge representation language based on linguistics, psychology, and philosophy, was developed by John Sowa [9]. These graphs can be applied to 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 were chosen for the ASPIN system because they readily represent the semantics of natural language and map directly into formal logic, which can be used for reasoning and checking. Conceptual graphs are finite, connected, bipartite graphs. Concepts and relations are the two types of nodes. Concept nodes can represent objects, behavior or attributes. Relations describe the role of concepts with respect to other concepts in a graph. Arcs connect concepts with relations.
3.1.1 Concepts and relations

The two types of nodes in conceptual graphs are concepts and relations. Concepts are nodes used to signify objects, behavior or attributes of a system. Relations are nodes which establish links between the concepts. Concepts are represented by rectangles and relations are represented by ellipses. Directed arcs connect concepts to relations and vice versa to show how two concepts are linked by a relation. Every concept has a label associated with it and the label has two parts namely the concept type and the referent for that particular concept. The referents depend on whether a concept is generic (represented by *) or individual (represented by # or a specific name). A relation node is also assigned a type label. The concept types and relations are predefined for a system.

Conceptual graphs can be represented in pictorial or textual form. An example of a conceptual graph in both forms is shown in Figure 3.1 & 3.2 with an English sentence.

*The processor reads data from the memory*

![Diagram](image)

*Fig 3.1 Pictorial conceptual graph*
\[ \text{read : * } \]
\[ \rightarrow (\text{agnt}) \rightarrow [ \text{processor : # } ] \]
\[ \rightarrow (\text{obj}) \rightarrow [ \text{data : * } ] \]
\[ \rightarrow (\text{src:from}) \rightarrow [ \text{memory : * } ] \]

**Fig 3.2 Textual conceptual graph**

From the pictorial notation it can be seen that \([\text{read : *}]\) denotes a generic \textit{read} concept. (The \textit{data} and \textit{memory} concepts are also generic). An individual marker in the processor concept's referent field is denoted by the pound sign '#'. This is to identify a particular instance of a concept. The pictorial form shows the root concept \textit{read} being connected to other concepts by appropriate relations. The \textit{processor} is the agent(\textit{agnt}) which does the \textit{read} action, the object(\textit{obj}) of the \textit{read} being \textit{data} and source(\textit{src}) being \textit{memory}.

The textual form shown below the pictorial form conforms to the notation in Sowa's book on Conceptual Structures [9]. Concepts are written within square brackets and relations are attached to the concept with arrows denoting the direction of the relation. The relations are enclosed within parentheses. The markers in the individual fields may be generic, individual or specific depending on the description of the concept in the sentence.

### 3.1.2 Concept type hierarchy

Concepts do not have any meaning in isolation. The semantics of a concept are defined by its relationships with other concepts. The concepts are ordered in a hierarchy. Starting,
with a universal type which encompasses all the concepts, the rest are subtypes of this universal type. Under each type of concept, there might be several subtypes. A partial hierarchy is shown in Fig 3.3. Every concept in the type hierarchy has a subtype(below) and a supertype(above) except for the universal type and absurd type. The absurd type is the lowest in the hierarchy and is a subtype of all the other types. The hierarchy is a lattice structure and every type in the lattice inherits the properties of the type above it.

![Type lattice in a conceptual system](image)

**Fig 3.3 Type lattice in a conceptual system**

### 3.1.3 Canonical graphs

Conceptual graphs are formed with a defined set of concepts and relations. Any combination of concept and relation may not lead to a graph which is meaningful though it might be correct according to the syntactic rules. Certain basis graphs are defined
restricting a concept to be connected to other concepts only in a particular manner in order to be meaningful. Such graphs are called Canonical graphs. A canonical graph is usually star shaped with the root concept connected to other concepts through appropriate relations. Every defined concept would have a corresponding canonical graph. Examples of canonical graphs are seen in Figure 3.4 for concepts read and data. The read is the root concept and has an agent which performs the read and an object which is read. The canonical graph for data illustrates that the data could be located in the memory.

![Diagram](image)

**Fig 3.4 Canonical graph for READ and DATA**

A set of canonical graphs is called a canon. Conceptual graphs are formed from the canonical graph set using rules referred to as canonical rules. New canonical graphs can also be developed from the existing canon.

3.1.4 Canonical rules

The canonical rules defined by Sowa to form new canonical graphs or conceptual graphs from the canon are *copy, restrict, join,* and *simplify.*
A canonical graph \( w \) can be derived from two existing graphs \( u \) and \( v \) as follows:

- **copy**: \( w \) is an exact replication of \( u \);
- **restrict**: a concept \( c \) in \( u \) may be replaced with a subtype of \( c \), and its referent may be changed to an individual marker;
- **join**: if a concept \( c \) in \( u \) is identical to a concept \( d \) in \( v \), then the two concepts may be merged, maintaining the existing links.
- **simplify**: if \( u \) has duplicate copies of a single relation the copies may be replaced with a single copy maintaining the existing links.

![Diagram](image)

**Fig 3.5** Canonical graph after restriction of value to data

Referring to Figure 3.4, the object of the \( read \), \( value \), can be specified to be \( data \) since \( data \) is a subtype of \( value \). This is an example of the restrict operation(Figure 3.5).

![Diagram](image)

**Fig 3.6** Two canonical graphs to be merged
In Figure 3.6, two identical concepts (data) in the graphs which can be merged are shown with a link (dashed line) between them. This link is called a co-referent link since the two concepts refer to the same entity. Two concepts which are found to be co-referent can be merged together. The graph obtained after the merge is seen in Figure 3.7. When two identical nodes are merged, the relations that are connected to all deleted copies of the node need to be connected to the retained copy of the node.

![Canonical graph of READ after Join](image)

**Fig 3.7 Canonical graph of READ after Join**

The canonical rules may be applied to simplify and integrate individual conceptual graphs into bigger graphs.

Concepts may have more than one canonical graph if used in different contexts. The appropriate canonical graph has to be chosen based on the context. An example of a concept with two canonical graphs is given in Figure 3.8. Here an identification method has to be established to select the appropriate canonical graph.
Another rule to be mentioned while retrieving canonical graphs from a database is the delete rule. This is not a canonical rule but has been used in the thesis work to construct a conceptual graph from the canon. The canon stores the canonical graph of a concept with all possible relations that the concept can accommodate. When a given graph is matched with its root concept's canonical graph, relations not found in the input graph are deleted from the retrieved canonical graph. This deletion simplifies the canonical graph and makes matching easier.
3.2 Semantics of Digital Systems

The set of concepts and relations used in conceptual graphs vary depending on the application domain. For any application domain, the concept and relation sets must be clearly defined. Concept types are mainly dependent on the entities in the domain. Relations that are defined should be adequate to represent all types of system behavior. In a digital system, the commonly encountered entities are devices or components of the system, actions describing the behavior of the system, values which are acted upon, events, and units which describe quantitative measures like time, frequency. The concept classification is based on these types as: **device, value, action, event, and attribute**. All hardware elements fall under **device** classification, software and data types under **value** type, operations and activities of the system under **action** type, memory and time measures under **units**. **Event** is a type defined to signify any discontinuous action. An **action** can be said to happen as a result of an **event** or be the cause of an **event** type concept. The table below summarizes these classifications.

**Table 1. Definition of Conceptual types**

<table>
<thead>
<tr>
<th>Conceptual Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>Any unit of hardware.</td>
</tr>
<tr>
<td>value</td>
<td>Any unit of information, software.</td>
</tr>
<tr>
<td>action</td>
<td>An activity or process, usually having a duration.</td>
</tr>
<tr>
<td>event</td>
<td>A discontinuity in one or more actions.</td>
</tr>
<tr>
<td>attribute</td>
<td>An attribute of the dimension or quantity of an object or action.</td>
</tr>
</tbody>
</table>
Once the concept set is defined, the type hierarchy has to be defined. The partial type hierarchy for the ASPIN system is shown in Figure 3.9. Each node in the hierarchy represents a concept type in the digital system domain. The hierarchy is a lattice, therefore any two concept types classified in a hierarchy have one least common supertype and one greatest common subtype. The Universal type is at one end of the lattice and serves as the supertype of all concepts. The types are divided into subtypes and at the lowest level of the hierarchy is the absurd type. The absurd type is a subtype of all the other types.

![Diagram of partial conceptual type hierarchy]

**Fig 3.9 Partial conceptual type hierarchy**

The conceptual relations specify the roles that the concepts play with respect to each other. Table 2 contains the list of conceptual relations used by the Model Generator and the definitions for them.
<table>
<thead>
<tr>
<th>Relation (abbreviation)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent (agt)</td>
<td>the device that performs the action.</td>
</tr>
<tr>
<td>and (and)</td>
<td>conjunction relation indicating subparts.</td>
</tr>
<tr>
<td>attribute (attr)</td>
<td>a characteristic of the object.</td>
</tr>
<tr>
<td>condition (cond)</td>
<td>the condition that enables the action or event.</td>
</tr>
<tr>
<td>contain (contain)</td>
<td>indicates the state of containment of an object by a compatible object</td>
</tr>
<tr>
<td>destination (dest)</td>
<td>the destination object of an action.</td>
</tr>
<tr>
<td>determiner (det)</td>
<td>a word that specifies an individual referent.</td>
</tr>
<tr>
<td>frequency (freq)</td>
<td>the frequency at which the action occurs.</td>
</tr>
<tr>
<td>instrument (inst)</td>
<td>the device to implement the action or event.</td>
</tr>
<tr>
<td>location (loc)</td>
<td>location of a stored value.</td>
</tr>
<tr>
<td>manner (man)</td>
<td>the manner in which an action is performed (adverb)</td>
</tr>
<tr>
<td>modal (mod)</td>
<td>an auxiliary that modifies a verb.</td>
</tr>
<tr>
<td>name (name)</td>
<td>the name of an object.</td>
</tr>
<tr>
<td>object (obj)</td>
<td>the concept that is the object of an action or event.</td>
</tr>
<tr>
<td>operand (opnd)</td>
<td>a value used by the operation.</td>
</tr>
<tr>
<td>or (or)</td>
<td>conjunction indicating a choice.</td>
</tr>
<tr>
<td>purpose (purp)</td>
<td>the purpose of device or action.</td>
</tr>
<tr>
<td>quantity (quant)</td>
<td>the number of objects.</td>
</tr>
<tr>
<td>result (rslt)</td>
<td>condition or value resulting from an action or an event.</td>
</tr>
<tr>
<td>size (size)</td>
<td>the size of an object.</td>
</tr>
<tr>
<td>source (src)</td>
<td>the source of an object of an action or an event.</td>
</tr>
<tr>
<td>type (type)</td>
<td>a description of the type of object.</td>
</tr>
<tr>
<td>value (val)</td>
<td>the value held by a device or a value.</td>
</tr>
</tbody>
</table>
3.3 Display

A display defined in the context of this thesis is a drawing consisting of a collection of display elements which has been generated from a conceptual graph. The elements of a display are icons, connectives, or groups of icons and connectives (a sub display). Each display element is produced by a call to a drawing procedure, and the entire display is produced by a script of drawing calls. Therefore, the terms display element and drawing call will be used synonymously. It may be desirable to edit the conceptual graph by graphically editing the display generated from it. In order to support this, it is assumed that every display element has a pointer back to the conceptual graph element which generated it.

To visualize the knowledge represented in a conceptual graph, a display is constructed using a set of icons and connectives appropriate to the semantic domain of the graph and the objective of the display. Icons may range from simple schematic symbols to highly parameterized pictures as might represent a textured solid object with shading. Similarly, connectives may be simple lines or may have constraints.

An icon is a picture element of fixed shape (type), but may vary in one or more size, color, or texture parameters, and may contain labels. For example, two rectangle icons may differ in aspect ratio and color, yet remain rectangles.

A label consists of one or more characters. Typical labels are concept and relation types, referent markers, and arc ordinals. In some cases it may be convenient to consider labels as rectangular, textured icons.
A **connective** is a line which connects two icons, and may have a texture or style attribute (e.g. solid, broken, invisible, arrow, plain). Connectives may also have labels. A connective may also have one or more constraints upon it as attributes. For example, a connective may be constrained to be vertical or to be of a length which is a function of the sizes of the icons which it connects. It is assumed here that the constraints are in the form of predicates. Examples of sets of icons and connectives for a digital system display types appear in Figure 3.3.

---

**Fig 3.10 Object set for the digital display scheme**
Fig 3.11 Literal display

Figure 3.12 shows a display corresponding to the graph in Figure 3.11.

Fig 3.12 Schematic display example
Chapter 4. Model Generator

The Model Generator [11] is the visualization tool which generates graphical displays from the input conceptual graphs. This chapter discusses the important features of the Model Generator and the algorithms involved in display generation.

4.1 Overview of the Model Generator

The Model Generator consists of two modules: a Mapping Engine and a Placer.

Fig 4.1 Overview of the Model Generator[11]
The Mapping Engine maps the input conceptual graph to a script, that generates the graphical representation. For each specific concept in the conceptual graph (CG), the Model Generator uses the concept type to retrieve a script for the canonical picture (CP) in the Interpretation Library. A script is a set of drawing commands required to create a graphic display for the concept. A graph index number is associated with every canonical picture for a concept type. This number is matched with the index number (gindx) of the concept in the input conceptual graph to identify the appropriate canonical picture. The generic nodes in the canonical picture are instantiated using the information in the input conceptual graph and by consulting the concept type hierarchy. The canonical picture script is then appended to the script assembled previously and redundant commands are removed.

The redundancies in the script can be: (i) multiple draw commands for icons for the same representation, or (ii) multiple draw commands for connectives between a pair of icons. 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 remaining script command. The containments list of the remaining command is modified to include the containments of the removed commands.

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. Sizing of each icon is performed using its list of containments and the size of its label string. This is a recursive process beginning 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 X-Window interface of the Placer 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.

4.2 Data Structures and notations

Chapter 3 introduced conceptual graphs and the linear/textual form of expressing them. This earlier form has been modified to suit the purposes of the Model Generator and Back Annotator. The notation used here attaches a unique identifier number to every concept. This is mainly to facilitate back annotation where the identifier can be used to associate every icon to its pre-image concept (the concept in the input conceptual graph which corresponds to that icon). Also every concept in the graph has a graph index number associated with it in its list of relations. The Mapping Engine uses this index number to match the correct canonical graph(picture) with the concept. In Sowa's notation, every concept has a type field and a referent field. The referent field is uninstantiated if the concept is a generic type and is marked with an asterisk (*) sign. In the Model Generator, the referent field has a subtype when the concept is generic. The label that is written
inside the icon in the display contains the concept type, the referent, if instantiated, and the concept identifier number (concept id). The modified notation for the sentence "The processor reads data from the memory" is:

\[
\begin{align*}
[1 : \text{read : read}] \\
\quad \rightarrow (\text{gindx : 1}) \\
\quad \rightarrow (\text{agnt : agnt}) \rightarrow [2] \\
\quad \rightarrow (\text{obj : obj}) \rightarrow [3] \\
\quad \rightarrow (\text{source : from}) \rightarrow [4] \\
[2 : \text{processor : processor}] \\
\quad \rightarrow (\text{gindx : 1}) \\
\quad \rightarrow (\text{det : the}) \\
[3 : \text{data : data}] \\
\quad \rightarrow (\text{gindx : 1}) \\
[4 : \text{memory : memory}] \\
\quad \rightarrow (\text{gindx : 1}) \\
\quad \rightarrow (\text{det : the})
\end{align*}
\]

In this notation the relations are attached to the concept identifiers instead of the concepts themselves as in the earlier notation. The relations also have a referent field in order to provide for specific relations described in the natural language description of the digital system. Fields are separated by colons. The graph index number is denoted by the relation \text{gindx} for every concept.
4.3 Canonical Scripts

Canonical scripts map the concepts and relations into appropriate display objects. The icon script line and the connective script line have slightly different formats. An icon script line specifies

1) shape of the icon
2) label to be written into the icon
3) position of the icon
4) dimensions of the icon
5) containments of the icon
6) the line index number for the icon

The script line for a connective specifies

1) the type of the connective
2) the label of the connective
3) the index number of the source icon
4) the index number of the destination icon
5) the index number for the connective

The format of a script line representing any icon is

[line number] shape ( label, coordinates, dimensions, {containments}).

The format of a connective script line is

[line number] shape ( label, source icon, destination icon, coordinates, {}).
4.4 The Interpretation Library

Canonical pictures are derived from the canonical graph for a concept. The canonical graph is transformed to a graphical display format and the script required to draw that display is stored in a file for reference by the Mapping Engine. Depending on the context, the same concept may have two or more different canonical graphs and scripts. An illustration of multiple canonical graphs is given for the concept "increment".

![Canonical graph for INCREMENT](image)

**Fig 4.2(a) Canonical graph (1) for INCREMENT**

![Canonical picture and script](image)

**Fig 4.2(b) Canonical picture and script(1) for INCREMENT**

Figure 4.2 shows the canonical graph which implies that the agent of the increment action is an event and the object that gets incremented is data. The canonical picture has a shaded icon which depicts an event being the cause of the increment action. To draw this picture, drawing commands are needed. The script lines are given along with the picture.
The other canonical graph shows the increment being performed by a device. In this case the agent relation is represented in the canonical picture by the increment (action) icon being contained in the device icon. The object of the increment action is value which is represented as a rounded rectangular icon in the picture. The script corresponding to this picture is shown beside the picture. When the mapping engine transforms concepts to drawing scripts, it adds script lines retrieved for a concept from the canonical script library (Interpretation Library) with the already assembled ones. The type hierarchy is also stored in a library classifying every concept. The Mapping Engine refers to the type classification and assigns icon shapes to the concepts based on their types. The retrieved
canonical picture is compared with the input information. The relations in the canonical picture for the concept which are not mentioned in the input graph are dropped.
4.5 Model Generation Algorithm

The Model generation algorithm is described in two parts with flow charts for the mapping process and the placement process.

4.5.1 The Mapping Algorithm

![Flow Chart of the mapping algorithm](image)

Fig 4.4 Flow Chart of the mapping algorithm
The input conceptual graph for the Model Generator is stored in a file and the file name is specified. The mapping process begins with parsing the input file and storing the concepts and relations in appropriate data structures. When a concept is read, a data structure is created and the input details are stored. Once parsing is over, the concepts are chosen one at a time and the script for the concept is retrieved from the Interpretation Library. The concept whose picture has been successfully retrieved from the library is marked. If the appropriate script is not found, then an error message is printed. The generic parameters in the script line are replaced with the actuals from the concept information. The retrieved script lines are attached to the earlier ones. If this gives rise to multiple occurrences of drawing calls for the same display element (redundancies) the repetitions are eliminated and the script lines are renumbered. When all concepts are found to be marked the final script for the display is ready to be passed on to the placer. The dimension and position parameters in the script lines are not yet instantiated by the mapping engine.
4.5.2 The Placer Algorithm

![Flow chart of the Placer algorithm]

Fig 4.5 Flow chart of the Placer algorithm [11]
The Placer's task is to read the script lines and allocate dimensions and positions to the display objects. The dimensions of the icons depend on their containments. In the case of an icon which does not have any other containment except for the label, the length of the label string is determined and the dimension is accordingly calculated so that the label can be fit inside the icon. For icons with containments, the size of the innermost icon is first decided and then the outer icons are sized accordingly. The positional parameters are calculated using a force-directed placement algorithm which models the display as a graph with vertices and edges. Initial positions are assigned for the icons. The forces of attraction and repulsion between the icons are calculated and translated to displacements from their initial positions. Icons which are linked by connectives are said to have a force of attraction and are placed closer. This calculation is performed iteratively and the base coordinates are decided for the outermost icons. Using these coordinates the position of the inner icons(containments) are calculated with respect to the outer icon. The connectives are assigned coordinates based on the center coordinates of the icons they link.

Once the placer has assigned coordinates to both icons and connectives, the Xlib drawing procedures are called with the parameters to draw the display. The Model Generator has been implemented in the C language in an X-Windows environment on a Sun workstation platform.
Chapter 5. Back Annotation

5.1 Overview of the Back Annotator

The Model Generator was developed to support validation of automatic interpretation of input English sentences. This validation support is provided by generating a graphical display (the interpretation) for the conceptual graph for each input sentence. The Back Annotator has been developed as an enhancement to the Model Generator to provide the user with graphic editing facilities enabling automatic editing of the conceptual graph if the automatic interpretation is incorrect.

In natural language, a sentence may have more than one interpretation. The automatic interpreter can choose any one interpretation. The Back annotator provides an option to modify the interpretation in the conceptual graph through the display.

The interpretations for the sentence shown below are given as an example for ambiguity in natural language specifications.

"The processor sends the status and outputs to the port"

This sentence can be interpreted in three ways: (1) the processor sends the status and also outputs to the port, (2) sending the data causes the processor to output to the ports, and (3) both status and outputs are values sent to the port. The automatic interpreter could have chosen the first interpretation while the specifications author meant the second or third one or vice versa. The displays for all the interpretations are shown in Figure 5.1
along with the conceptual graphs for the displays. Depending on which interpretation is suitable, the display can be modified from one to the other using the Back Annotator.

(a)

[1: move]
  -> (and) -> [2]
  -> (and) -> [3]
  -> (agent) -> [4]
[2: send]
  -> (obj) -> [5]
[3: output]
  -> (dest:to) -> [6]
[4: processor: #]
[5: value : *]
  -> (name) -> [status;]
[6: port : #]

(b)

[ 1 : send : send ]
  -> (agent) -> [ 2 ]
  -> (obj) -> [ 3 ]
  -> (dest : to) -> [ 6 ]
[ 2 : processor : # ]
[ 3 : and : and ]
  -> (obj: null ) -> [ 4 ]
  -> (obj: null ) -> [ 5 ]
[ 4 : value : * ]
  -> (name) -> [status]
[ 5 : value : * ]
  -> (name) -> [outputs]
[ 6 : port : # ]
5.2 The Back Annotator

The Back Annotator provides a graphical interface to modify a conceptual graph in the database. The user can edit the picture by adding, deleting, modifying or moving the objects on the display. Editing the pictorial notation is easier than editing the conceptual graph as the latter requires knowledge of conceptual graph theory. This section describes the editing operations in detail with examples.

5.2.1 ADD operation

When an interpretation in the database is found to be insufficient in detail, it can be elaborated by adding new objects to add more information. The Back Annotator provides
choices to add icons or connectives to an existing display. To add an icon, the user chooses the ADD operation and inputs the type of icon and an appropriate label to be attached to the new icon. The placement of the icon is selected by the user. In case of connective addition, the type of connective and its label are entered by the user. The source and destination icons of the connective are chosen in the display. In the add operation, the display is updated to reflect the changes specified by the user.

The Back Annotation for the addition of an icon to the display is addition of a corresponding concept. The concept name is derived from the label of the icon and a concept number is attached to the name. When a new connective is added it requires the addition of a new relation to the conceptual graph. The type of relation is decided on by the type of icons linked by the connective. Table 3 below shows a list of possible relations for different combinations of icons. Combinations of icons which cannot be linked by a connective are marked out. The Back Annotator gives a warning to the user when an illegal combination of icons is chosen by the user.
The choice of relations in the above table for different combinations of icons was made by studying a variety of conceptual graphs and the icon combinations that can arise in the Model generator.

New relations are introduced in the case of icons which are added inside already existing icons. The inner icons are termed as containments of the outer icon. Only certain combinations of containments are semantically meaningful. When an action icon is contained within a device icon, the device is the agent which does the action. The relation to be added to the conceptual graph in case of a containment is decided based on the type to which the inner and outer icon belong. Table 4 shows the list of relations for different combinations of icon containments.
Table 4 Relations for possible containments

<table>
<thead>
<tr>
<th>Icons</th>
<th>inner</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rect</td>
<td>circ</td>
<td>round_rect</td>
</tr>
<tr>
<td>rect</td>
<td>contain,</td>
<td>agent</td>
<td>location,</td>
</tr>
<tr>
<td></td>
<td>part</td>
<td>contain</td>
<td></td>
</tr>
<tr>
<td>circ</td>
<td>XX</td>
<td>and, or</td>
<td>XX</td>
</tr>
<tr>
<td>round_rect</td>
<td>XX</td>
<td>XX</td>
<td>XX</td>
</tr>
</tbody>
</table>

Example: "The data is sent"

<table>
<thead>
<tr>
<th>send</th>
<th>data</th>
</tr>
</thead>
</table>

[ 1 : send : send ]

  -> (gindx : 1)

  -> (obj : obj) -> [2]

[ 2 : data : data ]

  -> (gindx : 1)

  -> (det : the )

Fig 5.2 Display generated by Model Generator (before Add)

Figure 5.2 shows the initial display that is generated by the Model Generator for the given conceptual graph. To add more information to the existing conceptual graph,
the user chooses to add memory as the destination of the 'send' action. When an icon is added to the display, a corresponding concept has to be added to the conceptual graph. The concept is determined from the label given to the icon. From the label *memory*::3, a concept [3 : memory : memory] is added to the conceptual graph. The gindx is a relation that needs to be added for all concepts. The default index number is 1.

The modified display is shown below with the edited conceptual graph that corresponds to it.

![Conceptual Graph](image)

[ 1 : send : send ]
-> (gindx : 1)
-> (obj: obj) -> [2]

[ 2 : data : data ]
-> (gindx : 1)
-> (det : the )

[ 3 : memory : memory ]
-> (gindx : 1)

Fig 5.3 Display after Add (Step 1)

The next step of the modification is to add connective to the newly added icon from one of the existing icons. The relation added to the graph depends on the type of concepts connected by the relation. In this example, an action concept *send* and a device concept
memory are connected. The source is the action concept of type send and the destination is the device concept. So a relation (dest : to) is introduced between send and memory.

\[[ 1 : \text{send} : \text{send} ]
\quad\rightarrow (\text{gindx} : 1)
\quad\rightarrow (\text{obj} : \text{obj}) \rightarrow [2]
\quad\rightarrow (\text{dest} : \text{to}) \rightarrow [3]
\]

\[[ 2 : \text{data} : \text{data} ]
\quad\rightarrow (\text{gindx} : 1)
\quad\rightarrow (\text{det} : \text{the})
\]

\[[ 3 : \text{memory} : \text{memory} ]
\quad\rightarrow (\text{gindx} : 1)
\]

**Fig 5.4 Display after Add (Step 2)**

The modified display conveys more specific information than the original display. The conceptual graph is automatically back annotated.

Certain connective additions which do not map directly to simple addition of relations due to the non-isomorphic nature of the mapping from conceptual graphs to displays. This aspect of back annotation is explained in detail in the implementation section.
5.2.2 DELETE operation

Objects in the display can be removed using the delete operation. To delete an icon or connective, the user chooses the object after selecting the delete function. The display is updated after every delete operation.

To back annotate the deletion, the concept corresponding to the chosen icon is deleted. In case of connective deletion, the selected icons are mapped back to their corresponding concepts. Depending on the types of these concepts, an appropriate relation is removed from the conceptual graph. In case of containments when an inner icon is deleted, the relation indicating the containment relation between the two icons is marked for deletion. The relation deleted in this case also depends on the type of the inner and outer icons.

Example:
"the i80486 contains a cache and a co-processor"

Fig 5.5 Display generated by Model Generator (before Delete)
In Figure 5.5, the icon representing i80486 device contains the cache and co-processor icons. The figure below indicates the effect of a deletion on the display and the conceptual graph.
When an inner icon is removed, a concept corresponding to it and the relation which indicates the containment have to be deleted. The drawing scripts are modified by deleting the command corresponding to the removed icon and changing the containments list in the drawing command of the outer icon. To modify the conceptual graph, the concept and relation to be deleted are marked when the editing is done to the display and deleted while
writing the output (modified) conceptual graph. In the example output conceptual graph above, the co-processor concept and its relation gindx are removed. Also the reference to the concept number 5 in the relation set of concept number 3 implying containment has been deleted.

5.2.3 MODIFY operation

The Modify operation is used to change an existing concept type to another concept type. Restrictions apply, however. As in other editing utilities, the display is updated after the user completes a modify operation.

The modify operation can be used to modify a concept to another concept belonging to either its supertype or subtype. Modifying a concept to a supertype concept is termed generalization and modification to a subtype is termed specialization of a concept. As an example, a rectangle label can be changed between type *device* and *ALU* but not between *device* and *message*. This restriction has been imposed to restrict changes in the conceptual graph to a manageable level. Modifying a concept to one of a significantly different type can require many changes governed by a complex set of rules and requiring subgraph substitution.

In the conceptual graph, the modification is done by substituting the earlier concept name with the newly specified one. It is essential in this editing operation that the set of relations which existed for the earlier concept type should be valid for the new concept type. To ensure this, all relations of the modified concept are checked for validity with the
new concept. If there is a mismatch the user is warned. In case of changing the concept to a subtype most of the existing relations should be valid.

In conceptual graphs, there could be concepts which are attributes that qualify a concept. These are attached to the main concept by relations describing the nature of the attribute. In such cases, when an attribute field of an icon label is modified in the display, the concept representing the attribute in the conceptual graph undergoes modification.

The option to modify a connective has not been provided since it is simpler to implement the modification through deletion of the original connective and the addition of a new connective.
The chip has an on-chip 8K cache

[1 : have : have ]
   -> (gindx : 1)
   -> (agnt : agnt) -> [2 ]
   -> (obj : obj) -> [3 ]

[2 : chip : chip ]
   -> (gindx : 1 )
   -> (det : the )

[3 : cache_1 : cache ]
   -> (gindx : i)
   ->(loc : adj) -> [4 ]
   >(size : adj) -> [5 ]
   -> (det : an )

[ 4: physic_struct : on-chip]  
   ->(gindx : i)

[ 5 : mem_measure : 8k ]
   -> (gindx : 2 )

Fig 5.7 Display and conceptual graph before modification
The above display shows an icon corresponding to a chip containing another icon which symbolizes an on-chip cache. The editing done for this example is to change the label of the cache_1 icon to a different size. To illustrate, the label is modified to cache_1:on-chip:24k:3. This modification does not change the concept type but the attributes of the concept need to be changed. The concept type remains the same. The display is modified to show the change after the modify operation is completed by the user. For modifying the conceptual graph, the name and number of the original concept and the modified label are stored. These changes are implemented when the modified conceptual graph is written to the output file. The modified display and the conceptual graph are shown in Figure 5.8.
Fig 5.8 Display and conceptual graph after modification
5.2.4 MOVE operation

The Move operation allows the user to change the position of an icon on the display. Repositioning the objects could be to improve the aesthetics of the layout or to change containment relations between concepts in the conceptual graph. The user can move display objects by choosing the Move option from the menu and choosing the object to be moved.

In the conceptual graph the change effected depends on the Move operation. If an icon does not have any containment or is not a containment itself, the Move is done only to improve the placement of icons on the display. This requires no change in the conceptual graph. On the other hand if an icon is moved into another icon a new relation is introduced between the concepts corresponding to the inner and outer icons. Moving an icon out of an outer icon requires deletion of the relation between the two corresponding concepts. Table 4 shows the relations that will be affected for the icon combinations changed in the display. The relation involved in the operation depends on the type of the icons involved in the move.
"the register contains the 16-bit address"

[ 1 : contain : contain ]
  -> (gindex : 1 )
  -> (det : the )
  -> (agt : null ) -> [ 2 ]
  -> (obj : null) -> [ 3 ]

[ 2 : register : register ]
  -> (gindex : 1)

[ 3 : value : address ]
  -> (gindex : 1)
  -> (size : null) -> [ 4 ]

[ 4 : mem_measure : 16-bit ]
  -> ( gindex : 1 )

Fig 5.9 Display before move

This conceptual graph and the display above show the containment of a value icon in a device icon. The display indicates that the register stores an address. When the user moves the value icon out of the device icon the containment relationship is not valid anymore and has to be deleted from the graph. In the conceptual graph, the outer icon is
checked for a relation listed in the Table 4. If there is such a relation it is stored in the
delete list and deleted when the conceptual graph is written into the output file.
Alternatively, in the absence of a suitable relation, a concept of type contain or have is
searched for in the graph whose agent is the concept corresponding to the outer icon. The
contain concept has to be marked for deletion if the outer icon has no other containments
other than the icon which has been moved. In case of more than one inner icon, the
relation in the contain or have concept corresponding to the moved icon alone is marked
for deletion. In the example shown, the contain concept in the graph is completely deleted
to back annotate the change in the display.

\[
\begin{array}{c}
\text{register} : 2 \\
\end{array}
\]

\[
\begin{array}{c}
\text{value:address:16-bit:3} \\
\end{array}
\]

\[
[ 2: \text{register : register } ]
\]
\[
\rightarrow (\text{gindx : 1})
\]
\[
[ 3: \text{value : address } ]
\]
\[
\rightarrow (\text{gindx : 1})
\]
\[
\rightarrow (\text{size : null}) \rightarrow [ 4 ]
\]
\[
[ 4: \text{mem_measure : 16-bit } ]
\]
\[
\rightarrow (\text{gindx : 1})
\]

\textbf{Fig 5.10 Display after move}
5.3 Algorithms of the Back Annotator

The pseudo-code algorithms for the back annotator are given in this section. The modules are divided under separate titles here for each editing utility in the Back Annotator. The script buffer referred to in the algorithm is the set of drawing commands used to generate the graphic display.

5.3.1 ADD Algorithm

The Add algorithm describes the procedure for icon or connective addition. If an icon is being added then a check has to be made to see if the user is adding the icon inside another icon. Containments introduce new relations between the concepts represented by the icons. For every addition a new script line is added to the script set. This modified script set is used to redraw the picture on the display. The additions to the conceptual graph are also done.
Pseudo-Code:

begin
if (object = ICON)
{
    if (the coordinates chosen to add the new icon are close enough to the position of an existing icon)
    {
        Store the concept number of the closest icon;
        Construct the script line for the icon to be added from the type and label specified;
        Append script line to the script buffer;
        Search for the script line corresponding to the closest icon and add the script line number of the new icon to its containment list;
        Construct the concept line to be added to the conceptual graph and store it in the Add list;
        Introduce a relation between the new concept and the concept corresponding to the outer icon;
    }
else
    {
        Construct the script line for the icon to be added from the type and label specified;
        Append script line to the script buffer;
        Construct the concept line to be added to the conceptual graph and store it in the Add list;
    }
end if;
}
end if;
if (object = CONNECTIVE)
{
    Obtain concept number of the originating and terminating icons of the connective Add script line for a connective with appropriate source and destination icons from the earlier step;
    Add a new relation depending on the type of the concepts that are involved in the editing operations and store it in the add list;
}
end if;
Redraw display with the new contents of the script buffer;
end
While writing additions to the output file,

\[
\text{begin} \\
\text{Read input conceptual graph;} \\
\text{if (add list not empty)} \\
\{ \text{if (new relations are to be added to the concept read)} \text{ } \\
\text{Append the relation from the add list to the output conceptual graph;} \\
\} \text{endif;} \\
\text{if (end of input file reached)} \\
\text{Write all new concepts and relations attached to them to the output conceptual graph;} \\
\text{endif;} \\
\text{end} \\
\]

5.3.2 DELETE Algorithm

The Delete algorithm implements a delete operation on the display and back annotates it into the conceptual graph. A delete requires the user to select the icon to be deleted using the mouse. If a connective is to be deleted, the source and destination icons have to be indicated again using the mouse to identify the connective for deletion. The script is modified to reflect the deletion and the concepts and relation to be deleted are stored in a delete list. When all the editing operations on the display have been completed, the output is written to a new file. While writing the output file, the list of delete objects is read and selective concepts and relations are excluded from being written into the output file.
Pseudo-Code:

begin
if (object = ICON)
{
    Get icon which is closest to the selected coordinates on the display;
    Delete the script line corresponding to this icon;
    Delete the connective script lines which contain this icon as either the source or the destination;
    Store the concept and its relations to be deleted from the conceptual graph;
}
else
{
    Identify the connective selected for deletion from the source and destination icons;
    Mark the script line corresponding to this connective;
    Store the relation to be deleted corresponding to this connective by comparing the concept numbers involved in the deleting operation;
}
end if;
Redraw display with the new script buffer contents;
end

While writing to the output file,

begin
    Read input conceptual graph;
    if ( delete list not empty )
    {
        if ( concept or relation read from the input graph is marked for deletion )
            do nothing;
        else
            write the line read from the input graph to the output file;
    }
end
5.3.3 MODIFY Algorithm

The Modify algorithm modifies only a concept by changing the label of an icon. The new label for the icon is entered by the user. The label in the script line corresponding to the icon is modified. The new label is stored. From the identifier number, the concept to be modified is identified. The new concept name is stored to be written into the output file.

The display is redrawn with the modified script set.

\begin{verbatim}
begin
Modify the label in the script line by replacing it with the new label;
Identify the concept which is being modified;
If (main concept is modified)
{
   Mark the concept line;
   Store the modified concept name that would replace the old concept name;
   Check if existing relations are valid for this concept by comparing them with the relations for the type of the modified concept name in the interpretation library;
   if (any relation is invalid)
      display warning message to user;
}
else
{
   Check adjacent concepts which are related to the main concept to identify the attribute concept that was modified;
   Mark the concept line;
   Store the modified concept name that would replace the old concept name;
}

While writing to the output file,

begin
if (modify list not empty )
{
   if (concept read from the current line is marked for modification )
   {
      write the modified concept line to the to the output conceptual graph;
   }
   else
      write the line read from the input conceptual graph;
end
\end{verbatim}
5.3.4 MOVE Algorithm

The Move algorithm is utilized to move icons in the display. In this option, a check has to be done to determine if a Move operation by the user gave rise to or eliminated any containment relations.

\begin{verbatim}
begin
Get the icon selected for move operation;
if (the selected icon is a containment of another icon)
    { if (the destination coordinates are not close enough to the coordinates of the outer icon)
        remove the selected icon index number from the containment list of the outer icon;
        Mark the relation which describes the containment;
        if (no containment relation found)
            { Search for a concept which relates the two concepts corresponding to the inner and outer icons as containment;
              Mark the concept and its relations for deletion;
            }
    }
}
end;

if (destination coordinates are inside the boundary of another icon)
    { Get the index number of the icon that is being moved;
      Add the icon index number to the containment list of the script line corresponding to the outer icon;
      Determine the new relation to be added between the concepts corresponding to the icons;
      Store the relation string to be added to the conceptual graph;
    }
endif;
Redraw display with the contents of the modified script buffer;
end
\end{verbatim}

While writing the output conceptual graph, the delete or add algorithm is used to either delete or add concepts or relations depending on the effect of the move operation.
5.4 Implementation of the Back Annotator

The Model Generator was initially designed as a visualization tool for the conceptual graph knowledge base and was only an output window. To incorporate back annotation, the window has been modified to a input/output window. The user interface required to specify the editing options has been built into the front end of the tool. The existing internal data structures were used and new data structures have been created for storing the user specifications. These data structures are used while back annotating the edits into the conceptual graph. Each type of edit operation has a list of concepts or relations which need to be appended, deleted or modified when written to the output file.
A block diagram of the Back Annotator is shown in Figure 5.11 indicating the interaction between modules. The modules have been divided depending on their functions and portability.

In the Back Annotator, the front end module consists of the external graphic user interface (GUI module) with menus and graphic display. This GUI module serves as the user interface to the Back Annotator. The main event loop keeps checking for user input
(mouse clicks or keyboard presses called events) and passes it on to the GUI module. The GUI module associates a separate procedure for a set of selected events on the display window. These procedures termed call back procedures are based upon what is to be done to implement the editing utility chosen by the user input. The events that are to be checked for and their corresponding call back procedures are specified when the graphic user interface is built. The external graphic user interface of the Back Annotator has been developed using X-View toolkit [16], a tool used in building menu-oriented user interfaces. The GUI module is an XView dependent module since the objects which constitute the display and the interface have been developed specifically for the XView environment.

The second module, independent of the graphic user interface is the one which back annotates the changes to the display at the script level and the conceptual graph level. The inputs the user specifies can be choice of editing operation, types, labels and positions of objects on the display. The parameters for modification are passed on to the global information module from the call back procedures. The Back Annotator uses these parameters and implements the required editing operation. The internal data structures and the file operations are handled by this module as well. The Mapping Engine and the Placer are used to generate the graphic display for the input conceptual graph. When the user edits the display, the script lines are accordingly modified and passed through the Placer again to generate the updated display.

The Graphics Drawing Module uses the drawing commands in X-Library [14] to generate the display on the screen. The X-Library function calls [15] for drawing on the display window use the data passed on by the global information module. This graphics
module is also subject to change depending on the graphics package used to draw the pictorial representation.

The options provided in the Back Annotator were decided based upon the design of schematic editors which are used to model digital systems. Any graphics application would require basic manipulation of display objects involving addition, deletion, and moving of the objects in the drawing area. Certain graphic manipulation functions have been included due to the specific nature of the interpretation task to be performed. This led to certain design choices on the editing options.

The Back Annotator restricts the user to choice of the specific shaped icons and connectives shown in Figure 5.12.

![Icons and Connectives permitted in Back Annotator](image)

**Fig 5.12 Icons and Connectives permitted in Back Annotator**
Certain specific design choices were required to suit this application. The icon EVENT in Figure 5.12 is also used as a cause bar icon for indicating a cause relation meaning that one action or event 'caused' another action or event to occur. A cause relation is valid only when the cause bar is connected to two suitable icons (action or event). The Back Annotator looks for validity in the case of a cause bar by checking if there is a cause and effect associated with every added cause bar. The back annotation is done for addition of cause bar only when both control connectives have been added between the cause bar and the other icons.

On choosing any of the options, the sub window for specifying the inputs is opened. The choices have to be entered and confirmed. The display is updated so that the user can immediately visualize the changes that have been suggested.

<table>
<thead>
<tr>
<th>MODGEN &amp; BACKANN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD DELETE MODIFY MOVE QUIT</td>
</tr>
</tbody>
</table>

Fig 5.13 Display Window of the Back Annotator

The figure above shows the header of the display window which provides the user with the choice to select one of the editing options by clicking on the appropriate choice. On clicking one of the four options, another menu is popped up on the screen. This pop-up menu takes the specifications for the corresponding edit. The pop-up menus for each of the choice in the main display window are illustrated below.
Once the choices are entered in the menu, the back annotation proceeds on the algorithm for each edit operation described in Section 5.2.
6. Back Annotation Examples

This chapter concentrates on a detailed example for each editing utility. The display generated by the Model Generator and the edited display are shown with the examples.

6.1 Add Utility Example

Sentence:

Rise of INC signal increments the data.

The conceptual graph for the sentence above is

[ 1 : increment : increment ]
  -> (gindx : 1 )
  -> ( agnt : agnt ) -> [ 2 ]
  -> ( obj : obj ) -> [ 3 ]
[ 2 : rise : rise ]
  -> ( gindx : 1 )
  -> ( obj : obj ) -> [ 7 ]
[ 3 : data : data ]
  -> ( gindx : 1 )
  -> ( size : adj ) -> [ 4 ]
[ 4 : mem_measure : 8-bit ]
  -> ( gindx : 1 )
[ 5 : signal : signal ]
  -> ( gindx : 1 )
  -> ( name : null ) -> [ 6 ]
[ 6 : id : INC ]
The Model Generator generates the display shown in Figure 6.1. To make the specification more precise, an icon of type *device* with a label *register* is added. When the type and device choices are confirmed, the location of the icon is selected. The location of the register icon is specified at the same location as the data icon to indicate that the data is contained in the register. The label, location and icon number are used to write a new script line for the newly added icon. The generic format of a script line is

```
[ line #] shape ( label, base_x, base_y, length, height, inside # [ line # ] [NULL] #)
```

where `base_x` and `base_y` specify the location of the icon and inside list gives the containments of the icon corresponding to this script line. Based on this generic format, the script line for the newly added icon is

```
[ 10 ] rect ( register : : 7, 100, 100, length, height, inside # [ 9 ] [ NULL] # )
```

This new script line is appended to the script buffer. The display regenerated after the addition of the new icon is shown in Figure 6.2.

For back annotation, a concept of the form [ 7:register : register ] is derived from the label given to the icon. The concept number, name and label are instantiated in the add list. Since the added icon was implied to contain the data icon, a relation is chosen from the table of relations for containments (Table 5.2). The relation that is chosen is ( loc : in) and the relation  -> (loc : in ) -> [ 7 ] is added to the list of concepts/relations to be added and concept number 3 is attached to this entry to indicate the concept under which the relation is to be added.

The modifications are confirmed for the current display and the display window is quit. The input conceptual graph is written to the output file fully since there are no deletions in this example. The new relation is added to concept number 3. At the end of the input conceptual graph, the new concept is added to the graph and the output file is completed.
The output conceptual graph is given below:

[ 1 : increment : increment ]
  -> (gindx : 1 )
  -> ( agnt : agnt ) -> [ 2 ]
  -> ( obj : obj ) -> [ 3 ]

[ 2 : rise : rise ]
  -> ( gindx : 1 )
  -> ( obj : obj ) -> [ 7 ]

[ 3 : data : data ]
  -> ( gindx : 1 )
  -> ( size : adj ) -> [ 4 ]
  -> ( loc : in ) -> [ 7 ]

[ 4 : mem_measure : 8-bit ]
  -> ( gindx : 1 )

[ 5 : signal : signal ]
  -> ( gindx : 1 )
  -> ( name : null ) -> [ 6 ]

[ 6 : id : INC ]

[ 7 : register : register ]
  -> ( gindx : 1 )
6. 2 Delete Utility Example

Sentence

*the* 180486 *contains a cache, processor and co-processor.*

The conceptual graph for this sentence is

\[ 1: \text{contain : contain} \]

\[-\rightarrow (\text{gindx} : 3)\]

\[-\rightarrow (\text{agt} : \text{agt}) \rightarrow [2]\]

\[-\rightarrow (\text{obj} : \text{obj}) \rightarrow [3]\]

\[2: \text{i80486 : i80486}\]

\[-\rightarrow (\text{gindx} : 1)\]

\[-\rightarrow (\text{det} : \text{the})\]

\[3: \text{device : and}\]

\[-\rightarrow (\text{gindx} : 1)\]

\[-\rightarrow (\text{and} : \text{null}) \rightarrow [4]\]

\[-\rightarrow (\text{and} : \text{null}) \rightarrow [5]\]

\[4: \text{cache_1 : cache}\]

\[-\rightarrow (\text{gindx} : 1)\]

\[-\rightarrow (\text{det} : \text{a})\]

\[5: \text{co-processor : co-processor}\]

\[-\rightarrow (\text{gindx} : 1)\]

The Model Generator display is shown in Figure 6. In this example one of the inner icons is deleted and the back annotation for the deletion is explained in detail. The display after deletion is shown in Figure 6. 4.
Figure 6.3 Display before "Delete"
To back annotate the deletion to the conceptual graph, the script buffer is updated with the edit done to the display. The script line representing the deleted icon is deleted from the buffer. The label, concept number, and concept name are stored in a list for deletes to be done in the conceptual graph. The display is redrawn with the script buffer's updated contents. The edits are confirmed and the output conceptual graph is generated. While writing the output conceptual graph, the list of deletes is checked for every line written to the output graph from the input graph to check if it matches the concept or relation marked for deletion.

The delete list stores the concept number 4 and the concept name cache_1. Since the deleted icon is a contained icon, an appropriate relation has to be deleted. The inner and outer icon are both devices in this case. So a possible list of relation is searched for in the table of relations for specific combinations of inner and outer icon types. Relations part or contain match the type of icon combinations seen in the example. Since such a relation is not found between concept numbers 2 and 4, a concept indicating the containment is searched for in the input graph. The concept [1 : contain : contain] matches the search. The agent of the contains is the concept i80486. The object of the contain is concept number 3 which does not match the concept corresponding to the inner icon. The concept number 3 is again searched for a relation connecting it to concept number 4. The relation [and : null] -> [4] matches and is stored to be deleted when the output conceptual graph is written. Further, since the icon for cache is deleted the concept representing the cache along with its relations is stored in the delete list. This list is checked when the output conceptual graph is written in order to delete the marked concepts and relations.
The resulting output conceptual graph is shown below:

[ 1: contain : contain ]
  -> ( gindx : 3)
  -> ( agnt : agnt ) -> [ 2 ]
  -> ( obj : obj ) -> [ 3 ]

[ 2: i80486 : i80486 ]
  -> ( gindx : 1)
  -> ( det : the )

[ 3: device : and ]
  -> ( gindx : 1)
  -> ( and : null ) -> [ 5 ]

[ 5: co-processor : co-processor ]
  -> ( gindx : 1)
6.3 Modify Utility Example

Sentence:

A high LD signal causes the 8-bit data in the R1 register to be loaded into the peripheral device.

The conceptual graph for this sentence is

[ 1 : load : load ]
  -> ( gindx : 2 )
  -> ( agnt : agnt ) -> [ 2 ]
  -> ( obj : obj ) -> [ 3 ]
  -> ( dest : into ) -> [ 4 ]

[ 3 : data : data ]
  -> ( gindx : 1 )
  -> ( size : adj ) -> [ 5 ]
  -> ( loc : null ) -> [ 6 ]

[ 2 : signal : high ]
  -> ( gindx : 1 )
  -> ( name : null ) -> [ 7 ]

[ 4 : device : peripheral ]
  -> ( gindx : 1 )

[ 5 : mem_measure : 8-bit ]

[ 6 : register : R1 ]
  -> ( gindx : 1 )

[ 7 : id : LD ]

Figure 6.5 shows the display for this conceptual graph. The icon representing the signal is modified by giving it a new label signal:low:LD:2. This does not cause any change in the shape of the icon. But when the attribute of the concept signal is changed from high to low, the conceptual graph would be affected. The display after this label modification is seen in Figure 6.6. The script line for the modified icon is identified and the label field is
changed. This effects the change in the display when it is redrawn. For the conceptual graph, the concept number corresponding to the modified icon and the old and new labels of the icon are stored in a list for modify operations. When the edit operations are complete, the output conceptual graph is written. The modify list is checked when every line is written to the output conceptual graph. When concept number 2 is written out, it is found in the modify list. The difference between the old and new label is determined. The modification to the signal gives it a new state. The line in the input conceptual graph with the concept type is modified to reflect the edit. The modified concept line is [ 2 : signal : low ] and is written to the output graph instead of the earlier line for concept number 2. Finally, the output conceptual graph is as shown in the following paragraph.

[ 1 : load : load ]
  -> ( gindx : 2 )
  -> ( agnt : agnt ) -> [ 2 ]
  -> ( obj : obj ) -> [ 3 ]
  -> ( dest : into ) -> [ 4 ]

[ 3 : data : data ]
  -> ( gindx : 1 )
  -> ( size : adj ) -> [ 5 ]
  -> ( loc : null ) -> [ 6 ]

[ 2 : signal : low ]
  -> ( gindx : 1 )
  -> ( name : null ) -> [ 7 ]

[ 4 : device : peripheral ]
  -> ( gindx : 1 )

[ 5 : mem_measure : 8-bit ]
[ 6 : register : R1 ]
  -> ( gindx : 1 )

[ 7 : id : LD ]
Figure 6.5 Display before “Modify”
6.4 Move Utility Example

Sentence:

The CPU manipulates the data in the register.

The conceptual graph for the above sentence is:

[ 1 : manipulate : manipulate ]
  -> ( gindx : 1 )
  -> ( agnt : agnt ) -> [ 2 ]
  -> ( obj : obj ) -> [ 3 ]

[ 3 : data : data ]
  -> ( gindx : 1 )
  -> ( loc : in ) -> [ 4 ]

[ 2 : processor : cpu ]
  -> ( gindx : 1 )

[ 4 : register : register ]
  -> ( gindx : 1 )

The original display is shown in Fig 6.7. To illustrate the move, the “data” icon is moved out of the “register” icon and the modified display is seen in Figure 6.8. A containment is altered in the display and this will affect the relation between the two concepts representing the inner and outer icons. When the display is redrawn, the script line for the outer icon is altered so that the inner icon is not a containment and the updated script buffer is used to redraw the display. Since an icon has been moved out of another icon, an appropriate relation is chosen from the table of relations corresponding to containments (Table 5.2). The relation is either location or contain. Since the input conceptual graph does not have any contain relation between the concept numbers 3 and 4, the relation
location is chosen. The concept numbers corresponding to the inner and outer icons and the type of relation that is to be deleted are added to the delete list.

When the output conceptual graph is being written, the delete list is checked to see if the concept or relation to be written is in the list. When concept number 2 is written there is a match between one of its relations and the objects in the delete list. The relation that matches with the relation type in the delete list is ( loc : in ) -> [ 4 ] and the related concept number also matches the concept corresponding to the outer icon. This relation is not written to the output conceptual graph. The resulting output conceptual graph is as below:

\[
\begin{align*}
&[1 : \text{manipulate : manipulate}] \\
&\quad \rightarrow ( \text{gidx : 1} ) \\
&\quad \rightarrow ( \text{agnt : agnt} ) \rightarrow [2] \\
&\quad \rightarrow ( \text{obj : obj} ) \rightarrow [3] \\
&[3 : \text{data : data}] \\
&\quad \rightarrow ( \text{gidx : 1} ) \\
&[2 : \text{processor : cpu}] \\
&\quad \rightarrow ( \text{gidx : 1} ) \\
&[4 : \text{register : register}] \\
&\quad \rightarrow ( \text{gidx : 1} )
\end{align*}
\]

The essential difference between a move and the delete in the back annotation procedure is that a "move" results only in the deletion of relations whereas in the "Delete" utility a concept itself can get deleted if the corresponding icon is removed.
Figure 6.7 Display before "Move"
7. Future work

7.1 Graphical Capture System

The Back Annotator captures the modifications in the display generated by the Model Generator. Using the back annotation capability, this graphic interface system can be used as a schematic capture system. The user creates the display to input system specifications and the corresponding conceptual graph is generated by the Back Annotator. Starting with a blank display, icons and connectives can be added one by one and the corresponding concepts and relations are added in the conceptual graph. This feature facilitates a user less familiar with conceptual graphs to generate one through the graphic interface. The existing menu in the interface can be modified to include a menu of icons and connectives for the user to make his selection of the object to be drawn by the click of the mouse.

7.2 Back Annotation and Model Generation for Physical systems

In the work described here, the Back Annotator and the Model Generator have been specifically designed for digital system representation. In a similar manner, any system specification can be described in terms of conceptual graphs. Model Generation and Back Annotation can be applied to any such system using these tools. A well-defined set of concepts and relations to model the system is required and this set needs to be mapped to a corresponding set of icons and connectives. An example is a system of physical objects moved around by a robot arm [6 & 13]. The physical objects could be of different shapes and are mapped to concepts indicating their shapes. The relations between the objects describe their physical location with respect to each other or a reference point. The display can be edited to indicate the actions of the robot arm to add, delete or move objects on the display and these modifications can be back annotated into the conceptual graph describing the display using the Back Annotator.
### 7.3 Animation in Display

Another area for future exploration in the Back Annotator & Model Generator can be addition of animation effect to the display. Currently, when an object is selected for move, the move is displayed by erasing the object at its original location and redrawing it in the newly selected location. Instead the movement of the object from its source location to destination location can be displayed as a continuous movement from its initial location to its destination. This animation feature can be very effective for displaying movement of objects in a physical system by a robot arm.

To display animation, it is necessary to define a special type of script line which can call specific procedures to implement the action suggested by the user. The drawing command for animation has the form animate(index, action, {parameter list}, duration, mark). The parameters passed to the procedure identify the operands of the action and the destination points of the action. The duration specifies the time over which the animation procedure operates and the mark is to indicate if the animation is complete or incomplete [6]. In the associated procedure, the initial and final points of the object will have to be used to calculate a path from one point to the other. Several intermediate points are chosen on this path and the object is redrawn at these points and erased when moved to the next point in the sequence. The duration of the move can also be programmed into the move operation. The path and the number of intermediate points of the move will be accordingly selected to meet the duration time specification.

Script lines which time the execution of the animation script lines are also essential to coordinate actions if there are more actions happening in a sequence. These script lines are defined as condition and transition drawing commands. They are of the form:

(a) `cond(index, condition, {parameter list}, mark)`

(b) `trans(index, {antecedents}, {consequents}, delay)`
The condition command contains a Boolean expression in its parameter list. When this Boolean expression is satisfied, the condition is said to be satisfied and the command is marked. The antecedents in the transition command are animation and condition drawing commands. When a transition takes place, all marks are removed from its antecedents. Marks are placed in all the animation commands which are the consequents of this transition. The antecedents and consequents are derived from temporal or causal relations between actions described in the conceptual graph [6].
REFERENCES


10. Sukthankar, S., Ramachandran, K., and Evens, M., Graphical User Interfacing with Domain Visualization Intelligent Medical Tutoring System, *Sixth Annual IEEE Symposium on Computer Based Medical Systems*


APPENDIX - USER'S MANUAL FOR BACKANN & MODGEN

The Back Annotator and the Model Generator provide the capability to graphically display the knowledge contained in a conceptual graph and modify if necessary.

The Back Annotator writes the back annotated conceptual graph to an output file and this file can be specified at the command line. To invoke Back Ann & Modgen type

```
modgen [-i <input file>] [-o <output file>]
```

The input file contains the conceptual graph in the textual form. This is the original input conceptual graph. The output file contains the modified conceptual graph in the same textual form.

Another choice than specifying an input file and output file explicitly for back annotation is to use a set of example conceptual graphs in the file mgsuite.log. The Model Generator is invoked by the command

```
modgen
```

at the shell prompt. The following set of choices appear:

1. quit
2. from file
3. mgsuite.log

The display for the input conceptual graph is generated. The display window has options to choose one of the edit operations from the menu in the display window. The steps to be followed by the user for each edit operation is explained in this section.
The first option quits the model generator. The second option takes the input conceptual graph from the file specified by the user and the file name is requested following the user option. The third option chooses the standard input file named mgsuite.log from which conceptual graphs can be viewed graphically either from beginning to end sequentially or individually by entering the number of the particular sentence that needs to be displayed. The display generated from the conceptual graph can be edited to add more details. The back annotated conceptual graph is written to a default output file named backann.out.

(i) ADD

a) Choose Add option on the display window by clicking on the menu item. A subwindow appears to specify a choice of adding an icon or a connective.

b) Choose between the icon and connective menu buttons. A new window appears to specify the type of icon/connective and the label that is to be attached to the object.

c) Enter type of icon/connective and the label. The label entered here should consist of two parts with the format type: label. Type and label should be separated by a colon. Label is specified in cases where there is a specific label to be assigned else it is the same as the type.

E.g. (a) cache: internal; (b) register: register

d) Confirm the entries by clicking on the “OK” button. If choice is incorrect, choose “CANCEL” button to erase the old entries.

e) If earlier entries are canceled, start from choosing Add option to specify a choice if an object needs to be added to the display. Once the editing is done, the user needs to specify the position of the icon on the display in case of an icon addition.

f) Specify the position of icon by clicking at the desired position of choice with the left most mouse button.
To add a connective, the source and destination icons will have to be specified.
g) Click on the source icon first and then the destination icon with the left mouse button.
The display is updated with the addition of the object and the type and label of the new object are stored for back annotation.
h) The addition of the concept “carrier” is a special case. Though “carrier” occurs as a concept in the conceptual graph, the display represents the carrier as a connective. Hence the carrier has to be specified as a connective with the type being carrier and the label appropriate to what the carrier in the particular case is. An example of a label for a carrier connective is given below.

E.g. carrier : bus : 16-bit

(ii) DELETE
a) Choose Delete option on the display window by clicking on the menu item. A subwindow appears to specify a choice of deleting an icon or a connective.
b) Choose between the icon and connective menu buttons.
c) Confirm the choice by clicking on the “OK” button.
d) Select icon for deletion by clicking on the icon itself. For deleting a connective, click on the source and destination icons of the connective.

(iii) MODIFY
a) Choose Modify option on the display window by clicking on the menu item. A subwindow appears with a prompt to enter the new label. If any of the attribute fields of an existing label is being modified then it has to be entered in exactly the same format as
the existing label. A new label (but concept type being the same) is specified in the format shown in the Add Section of this manual.

b) Confirm the choice of label by clicking on the “OK” button.

c) Select the icon which is to be modified by a click of the left mouse button.

(iv) MOVE

a) Choose Move option on the display window by clicking on the menu item. This edit utility also has only the choice of moving icons.

b) Confirm the option to move an icon by clicking on the “OK” button of the Move subwindow.

c) In the case of moving a single icon, choose icon to be moved by clicking the left mouse button.

d) If the icon has containments and has to be moved completely with its containments click the middle mouse button on the icon to be moved.

A common note for all the options above is that while selecting an icon it is best to click the mouse button close to the center of the icon.
Vita

Shreerekha Balachandar was born in Tiruchirapalli, India on the 22nd of July, 1971. She completed her Bachelor’s degree in Electronics and Communication Engineering from PSG College of Technology, Coimbatore, India in May 1992. She joined Virginia Tech in Fall 1992 to pursue a Master’s Degree in Electrical Engineering and concentrated in the areas of VLSI and digital system design. She is currently employed as an R&D CAD Engineer with Cirrus Logic, Inc. in Fremont, California.

Shreerekha Balachandar