INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A constraint-based 2-dimensional object display system

Lee, Sungkoo, Ph.D.
Case Western Reserve University, 1991
A CONSTRAINT-BASED 2-DIMENSIONAL OBJECT DISPLAY SYSTEM

by

SUNGKOO LEE

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Thesis Advisor: Dr. Gerald M. Radack

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY
January 1991
CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

Sungkee Lee

candidate for the Ph.D.
degree.*

Signed: Geraldo Radack
(Chairman)

Date Dec. 17, 1990

*We also certify that written approval has
been obtained for any proprietary material
contained therein.
I grant to Case Western Reserve University the right to use this work, irrespective of any copyright, for the University's own purposes without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

[Signature]
A CONSTRAINT-BASED 2-DIMENSIONAL OBJECT DISPLAY SYSTEM

Abstract

by

SUNGKOO LEE

This thesis will discuss a system for automatic layout and display of 2-D objects.

Our goal is to be able to display data of executing programs. We want the system to be usable with common higher-order languages such as C, and to require no modification of the source program.

We give the textual descriptions of the special objects with some rules or constraints and we derive several layout algorithms from the textual descriptions for how they should be displayed on the screen. Constraints are a useful tool to represent the relations that must be maintained between objects. They can be used to specify relations between objects, to maintain consistency between user program data structures and graphical depiction of objects, and to specify layout.

This system can be used to quickly create visual displays of information in a program, and can be applied to a visual debugger or some graphical drawing system, and has been implemented on a Unix systems in Prolog and C languages.
To my parents and my wife Helen
Acknowledgments

First and foremost, I would like to thank Dr. Gerald Radack for guiding and helping me with his invaluable knowledge in this area. Without his strong support, this work would not have been possible. I am especially thankful to Dr. George W. Ernst, Dr. Christos A. Papachristou, Dr. Francis Merat and Dr. Khin Yin for being on my dissertation committee and helping me through the various phases of my stay at Case.

I would also like to thank Dukki Chung, Djamal Belhandouz, and Nandita Mukhopadhyay for helping me with the implementation of system, and Gwang Hoon Park, Youngsuk Chi and Mr. Vincent E. Hilaire for helping me with writing of dissertation. I am especially thankful to Mr. Paul Pak for encouraging me to accomplish this work.

As a final acknowledgment, I would like to thank Admiral Chong Ho Kim, Chief of Naval Operations in The Republic of Korean Navy, for giving me the chance of education in the USA.
Table of Contents

Abstract ii
Acknowledgements iv
Table of Contents v
List of Figures x

1. Introduction
1.1 Overview 1
1.2 Motivation 3
1.3 Problem 4
1.4 Approach to Solution 5

2. Literature Review
2.1 Data Structure Description 6
2.2 Visual Debugger and Data Structure Drawing System 7
2.3 Graphical Drawing System 9

3. Graphical Description Language
3.1 Overview 11
3.2 Literature Review 11
3.3 Basic Requirements of a Graphical Description Language 14
3.4 Design of a Graphical Description Language 14
3.4.1 Primitive Objects 14

3.4.2 Basic Relationships Between Primitive Objects 15
3.4.3 Composition Operations 18

4. Constraints

4.1 Overview 20
4.2 Literature Review 20
4.3 Description of Constraints 22
4.4 Constraint Satisfaction 23
4.5 Approach to Constraint Solving 24
4.6 Example 26

5. Data Extraction

5.1 Overview 30
5.2 Data Structure Representation 32
5.2.1 Description of FigEl 32
5.2.2 Data Structure of FigEl 33
5.3 Symbol Table Description Language 35
5.3.1 Overview 35
5.3.2 Syntax of the Symbol Table Description Language 36
5.3.3 Typical Example 38
5.3.3.1 Single Variable 38
5.3.3.2 1-Dimensional Array 38
6. Layout Algorithms

6.1 Overview

6.2 General Layout Algorithms
   6.2.1 Atomic FigEl
   6.2.2 Composed FigEl
   6.2.3 Disconnected FigEl

6.3 Special Layout Algorithms
   6.3.1 Linked Lists
   6.3.2 Binary Trees
   6.3.3 M-ary Trees
   6.3.4 Graphs
   6.3.5 Routing Algorithm

6.4 Graph Layout Algorithm
   6.4.1 Introduction
   6.4.2 Previous Works
   6.4.3 Graph Layout Algorithm
      6.4.3.1 Overview
      6.4.3.2 Terminology
      6.4.3.3 Graph layout Algorithm
      6.4.3.4 Node Selection Method
7. Implementation

7.1 Overview
7.2 GDB
  7.2.1 Gdb Features
  7.2.2 Symbol Table Structure
  7.2.3 Mechanism for Adding Command
7.3 Data Extraction
7.4 Symbol Table Description Language
7.5 Construct FigEl
7.6 Graphical Description Language
7.7 Constraints Solving in Prolog
7.8 X-window Interface
  7.8.1 X-window
  7.8.2 Interface
7.9 Display
8. Conclusions

8.1 Work Completed 86
8.2 Future Work 87

References 89
List of Figures

3.1 Edge crossing of edge1(V₁, V₂) and edge2(V₃, V₄) 16
3.2 Vertex crossing of vertex V₁ and V₅ 17
3.3 Inside edge touch 17
3.4 Inside edge partial touch 17
3.5 Vertical_jcin(O₁O₂, Above) and horizontal join (O₁O₂, Right) 19
3.6 Place_inside(O₁O₂, Outside) and place_vertical (O₁O₂, Right, D) 19

4.1 Binary tree 25
4.2 Initial random place with C₁, C₂ 27
4.3 With C₇ 28
4.4 With C₄C₅ 28
4.5 Add node c with C₂C₄C₅ 28
4.6 With C₃C₆ 29
4.7 With C₇ 29

5.1 Overview of the system 31
6.1 Example of atomic figure 47
6.2 Example of composit figure 48
6.3 Placement 49
6.4 Linked List 50
6.5 Binary tree by the default layout algorithm 51
6.6 Binary tree 52
6.7 Rectangular grid 60
6.8 Sample graph 64
6.9 Adjacency matrix of $G$ 65
6.10 By random selection 65
6.11 By algorithm 66
6.12 Average distance / lookahead 66
6.13 Average time / node 67
6.14 Average time / lookahead 67
7.1 Overview of implementation 70
7.2 Sample of the Symbol Table Description Language 78
7.3 Sample of FigEl 79
Chapter 1

Introduction

1.1 Overview

This thesis describes the design and implementation of a system for automatic layout and display of 2-D objects.

Individual objects are described in terms of a language and this language is used to specify how the objects should be displayed on the computer screen. This system also incorporates a mechanism which uses constraints.

Our goal was to be able to display data of executing programs. We wanted the system to be usable with common higher-order languages such as C, and to require no modification of the source program. The symbol table of a debugger provides a convenient mechanism for interpreting the contents of memory as data structures. In order to build a more general system, a separate language was designed for the description of the symbol table. In this thesis this language is called a symbol table description language. The user can then specify a translation between the symbol table description language and a language describing the graphical elements called a graphical description language.

Data extraction is the process of converting the user's data structures of 2-D objects in the runtime environment into standard form, such as a
symbol table description language, which can be used for an object's drawing and display. An algorithm called a data extraction algorithm is developed for extracting the data from the program symbol table. From the symbol table description language we can create the standard data structures of 2-D objects, and these are mapped into the graphical description language for drawing of 2-dimensional objects on the screen. In order to do the mapping efficiently, a mapping language must be created.

We give the textual descriptions of the data structures, or engineering diagrams, with some rules or constraints, and we derive several layout algorithms from the textual descriptions for how they should be displayed on the screen.

Constraints are a useful tool to represent the relations that must be maintained between objects. They can be used to specify relations between objects, to maintain consistency between user program data structures and graphical depiction of objects, and to specify layout. A constraint can be represented by both a declarative description of the relation and a set of procedures for making that relation hold.

The following were taken into consideration during the design of the layout algorithm. First, the display size of the figures. Large objects are more readable, though the larger the object, the smaller the amount of data that can be shown on one screen. Secondly, the distance between individual objects, and finally, how to represent the relationships among the individual objects.
The sequence of the system's operation is: (1) data extraction from applications such as data structures or engineering diagrams; (2) mapping into symbol table description languages; (3) creation of the standard data structures of 2-D objects for drawing; (4) mapping into graphical description language; (5) layout of the drawing, and (6) picture drawing and display.

This system can be used to quickly create visual displays of information in a program, and can be applied to a visual debugger or some graphical drawing system, and has been implemented on a Unix systems in the Prolog and C languages.

1.2 Motivation

With the advent of inexpensive graphical workstations, high resolution graphical data displays using window systems are now available to the average user. Thus, a graphical display system which can be easily used should be developed. Such a graphical display system would be able to draw 2-D objects as well as able to be used in visual debuggers, and be instrumental in software development and education.

With the increases in computer speed, a vast amount of effort has been dedicated to software development and maintenance. A system which could easily draw information on a visual display would reduce software development and maintenance time considerably. In addition, this system could be used in visual debuggers and graphical drawing systems, and stretch the envelope of computerization.
We need to allow the user to customize the way drawings are made. The use of constraints to describe two-dimensional objects and creating layout algorithms would enhance consistency over current drawing methods, and with enhanced layout algorithms, a more efficient two-dimensional display system can be designed, which could be used in fields such as automatic data structure drawing and engineering diagram drawing.

1.3 Problem

Layout Algorithms Development

There have been many proposals for algorithms for the drawing of two-dimensional objects. There are not many problems in the case of simple objects. However, in the case of graphs, the layout algorithms have been designed only for specific cases, and especially in the case of automatic drawing on a computer, a new algorithm is needed. In addition, the time complexity is an NP-hard problem, and drawing time must be taken into consideration.

Constraints Solving

Constraints are basic tools to represent the relations that must be maintained between objects. They can be used to specify relations between objects, to maintain consistency between user program data structures and graphical depiction of objects, and to specify layout. To do these, we must
develop the methods to represent constraints and the methods to solve constraints. In addition, the time complexity must be considered.

**Customization**

A prototype 2-D object display system which can, with modifications made only to the mapping function, work with any debugger for the purpose of visual debugging is needed.

1.4 Approach to Solution

To realize the system mentioned above, we need to follow three steps.

First, we should define the relationships between objects with constraints, and how the objects should be drawn and displayed with the constraints. To do this, we define the primitive objects, and the relationships between them, and find out how to solve these relationships using constraints.

Second, we have to develop several languages for description of the symbol table, object display, and mapping languages for mapping between them. To do this, we define the standard format of the symbol table for display and develop a language for this, and develop a language describing the graphical elements.

Third, we should develop or enhance the several algorithms for data extraction from the system symbol table and layout for drawing and displaying.
Chapter 2

Literature Review

2.1 Data Structure Description

Since the system automatically displays program data structures, it is important to have a formal understanding of data structures. There are at least two levels at which program data structures are seen:

- Physical data structures are those directly supported by the programming languages.

- Abstract data structures are those used by the programmer and may not correspond directly to the physical data structures. For example, a tree can be represented by using an array or records with pointers.

Abstract data structures are well defined by Berztiss [Berz 75], Writh [Writ 76], Horowitz [Horo 80] and Harary [Hara 73], etc. However, only generally abstract data structure types are defined.

The aesthetic definitions of tree drawings can be seen in Wetherell and Shannon’s [Shan 79], and definitions of general data structures can be seen in Radack and Mateti [Rada 86]. This paper deals with the definition, display, and layout methods of the data structure diagrams. In a later paper Ding and Mateti [Ding 89] show the aesthetic rules of each type of data structure. The description of special data structure diagrams can also be seen in B-Tree
[Blac 75], Warzecha [Warz 87], Batini [Bati 85], Gait [Gait 85], Reingold [Rein 81], Tamassia [Tama 86], and Woods [Wood 82].

2.2 Visual Debugger and Data Structure Drawing System

A major use for an automatic drawing system is for visualization of program data structures, perhaps for debugging or teaching purposes. Most data structure drawing systems have been designed as subsystems of debuggers and are operated basically with a debugger.

Some time ago, Incense [Myer 83] was developed as a system for the visual examination of data structures in interactive debugging. Incense used a special language called Mesa, which could draw arrays, records and graphs, and was readily customized with the addition of a small amount of source code. It could generate pictures of many data structures during the execution of an actual Mesa program. It supported editing of the data structure’s display and allowed modification of the actual values stored with the help of a pointing device and keyboard.

Ojeda's DDS [Ojed 85] was a system developed for the display of data structures for interactive debugging. DDS was written in Pascal and was used for the debugging of Pascal. A special processor gathered the variable type information from the user's Pascal program. Then, for every variable declared, some lines of the code were inserted in the user program, and a new program was produced.

Warzecha's DSD [Warz 87] was designed for Pascal programs. A processor created a modified program with special action calls at the
beginning and end of each block. These function calls served to add or remove some line from a symbol table maintained by DSD. This symbol table was used to draw the pictures. Thus, DSD was independent of the implementation of Pascal. This interfered with debugging since the source program was changed. Customization was accomplished through the use of a special dsdmap file which was used to hold the information on the data structure diagram.

Desai's Akrti [Desa 88] is a data structure drawing system which was developed as an extension of the arcturus Ada [Will 84] programming environment. Desai's system uses Ada procedure calls for drawing; uses panning, zooming, and clipping; and displays the results onto an X window system.

Garg's xgdb1 [Garg 89] and a later enhancement system [Rada 89], were developed as a subsystem of the graphic debugger gdb, written in C, and run on a Unix system. The information on the data structure drawings is extracted from the gdb symbol table and displayed on the X window system. A later enhancement system has more features than xgdb1, such as a figure description language, and special layout algorithms. This system is closest to a customizable system.

There are also systems which are used to draw certain types of data structures.

Edger [Fran 89] is a system which is used to draw graphs. A graph description language was developed for this system. This system uses the subgraph to draw low levels of graphs.
Reingold's system [Rein 81] is the system for producing tidy drawings of trees.

Tamassia's system [Tama 88] is a graph drawing system. This system shows how to increase the readability of a diagram using automatic tools.

Carpano [Carp 80] is a system for the automatic display of hierarchized graphs for computer aided decision analysis. Carpano uses the relationships between each of the elements to draw the graph.

2.3 Graphical Drawing System

There have been many proposals for graphical drawing systems. Most of the systems which are being developed today are done using constraints via object-oriented methods. This section discusses constraints and object-oriented systems.

The first constraint system was Sketchpad [Suth 63]. This was a general purpose system for the drawing and editing of pictures on a computer. Sketchpad allowed the user to define new kinds of pictures by composing primitive picture types such as points and line segments. Constraints were used to specify conditions that the picture had to satisfy.

ThingLab [Born 79] was influenced by Sketchpad and was notably expanded in numerous ways. For instance, Sketchpad's domain of constraints included only geometric objects, while Thinglab included numerical objects as well as non-numerical objects. Objects were created, edited, and animated through a combination of menu selections and direct manipulation. Thinglab was implemented using the object-oriented language Smalltalk [Inga 78].
Thinkpad [Rubi 85] is a constraint based system, which is a graphical system for programming by demonstration. Thinkpad uses constraints for several aspects of the definition of a data structure, such as to provide strong typing, to denote the graphical hierarchy, and to express dependencies and relationships. Thinkpad provides a dynamic programming language based on data abstraction. A programmer first designs data structures by drawing appropriate graphical representations.

PBS [Will 86] deals with the problem of creating and manipulating data structures for applications using computer graphics.

Vlissides [Vlis 88] implemented a two-dimensional structured graphics library that is object-oriented in its implementation and presents an object-oriented model to the programmer. Structured graphics packages allow elements in a display list to be lists themselves, making it possible to compose hierarchies of graphical elements.

InterViews [Link 89] was developed at Standford University with the object-oriented language C++. InterViews provides a standard way to handle scrolling, zooming, and panning operations, and provides objects that can be structured hierarchically using composition objects that support a variety of layout styles.

Borning [Born 81], Jacob [Jaco 85], Moriconi [Mori 85], and Reiss [Reis 86] are all examples of graphical programming systems. These systems provide iconic interfaces for creating, editing, and animating programs.
Chapter 3

Graphical Description Language

3.1 Overview

There is a need for an adequate graphical description language when a system for the automatic layout and display of 2-D objects is implemented. A graphical description language is used to specify how the object should be displayed on the computer screen. The design goal of this language is such that the graphical description language sufficiently provides a textual description on the objects as well as provide information on how that data is to be displayed. In addition the language must be flexible.

3.2 Literature Review

A visual language is defined as a language which processes visual information. Chang [Chan 86] classifies visual languages into four categories: (1) Visual languages for actually programming with visual expression. (2) visual languages for supporting visual interactions, (3) visual languages for processing visual information, and (4) visual languages that deal with visual objects and are themselves visually represented. In this thesis, visual programming languages are the main focus of discussion.

When designing a visual programming language, one of the first obstacles is objects which do not have a implicit visual representation. These
objects include data structure types such as arrays, records, and trees. These abstract objects must be visually represented in a graphical manner in order to create an environment for information transfer between the computer and the user. The uses of these visual languages are apparent in the fields of computer graphics, various computer-user interfaces, as well as visual debuggers.

Visual environment area of the visual languages are PECAN [Reis 85], SDAM [Hero 80], etc.

PECAN was developed at Brown University and was designed to provide the user with multiple displays that concurrently show different aspects of the program. This system gives the graphical support for the visualization of a program. For example, if the program is represented as an abstract syntax tree, the user can give a hint for displaying a tree. As a program development and testing tool, PECAN provides a wide range of languages.

SDAM was developed at Computer Corporation of America and deals with visualization of data or information. It is used for the graphical expression of a relational database.

Figure description areas of visual language are IDEAL [Van 81], GDL [Newb 89], etc.

IDEAL is a graphics typesetting language in which two-dimensional figures can be expressed. The building block called a "box" is used to share some features with procedures and some with records in general-proposal programming languages and includes a system of constraints that declares the
relative position of its significant points. The language is intended to work with existing text formatting systems so that the text and figures can be typeset at the same time.

GDL is a graph description language which provides a high-level description of a graph and how it should be displayed. The format of a GDL description is composed of three parts: attributes describing the graph, the nodes and the edges. This language represent the low levels of a graph as subgraphs.

Object oriented constraint-based visual languages are ThingLab [Born 81], Coral [Szek 88], GROW [Bart 87], etc. These languages were developed by using the current trend of object oriented programming.

ThingLab was designed and implemented with extensions to Smalltalk and based on the idea of objects that communicate by sending and receiving messages. Objects’ descriptions and computational method are organized into class.

Coral is a constraint-based, object-oriented relational language and a new user interface toolkit. Coral supports defining constraints in the abstract, and then applying them to different object instances. Coral is implemented using the CommonLisp Object System and runs on the X window manager [Sche 86]. The classes define each kind of graphical object.

GROW is written in Interlisp-D, uses the object-oriented language Strobe [Smit 83], and runs on Xerox 1100 series workstations.
3.3 Basic Requirements of Graphical Description Language

The textual description of primitive graphical objects (e.g. lines, boxes, rectangle, arrows, etc.) must be possible. Also, composition operations such as joining, aligning, touching, and overlapping are included.

The graphical language is designed to be flexible, so that extensions can easily be added, and the many possible applications must be kept in mind.

Application data structures are displayed on the computer screen using data abstraction hints in coalition with the layout algorithm.

Each graphical object is considered to be included within a certain shape such as a box, a circle, etc. These can be labelled directly, or through the use of indices.

3.4 Design of Graphical Description Language

3.4.1 Primitive Objects

Primitive objects are the fundamental units of screen drawing and include elements such as dots, lines, rectangles, triangles, ellipses, and arrows.

Vertices or points are denoted as \( V_i \) and represented by x and y coordinates. It is then an easy task to describe primitive objects.

Straight lines and edges are represented by two vertices or one vertex with length and direction. Let \( D_i \) denote direction represented by degrees and \( L \) denote length then edge \( (V_1, V_2) \) or edge \( (V_1, D_1, L) \).

Boundaries of boxes are represented by four straight lines or four vertices. Let \( E \) denote edges then box \( (V_1, V_2, V_3, V_4) \) or box \( (V_1, V_2) \).
edge \((V_2,V_3)\), edge \((V_3,V_4)\), edge \((V_4,V_1)\) = box \((E_1,E_2,E_3,E_4)\). Lists are represented as a sequence of elements.

A triangle is represented as three straight lines or as three vertices. Representation is triangle \((V_1,V_2,V_3)\) or triangle \((V_1,V_2)\), edge \((V_2,V_3)\), edge \((V_3,V_1)\) = triangle \((E_1,E_2,E_3)\).

A circle is represented by center \(C\) and radius \(R\) as circle \((C,R)\), while ellipses include a parameter \(A\) to represent the variant and are represented as ellipse \((C,R,A)\), where \(A \leq 6.28\).

Arrows are represented as a straight line terminating at a head for arrow \((V_1,V_2)\).

3.4.2 Basic Relationships between Primitive Objects

A constraint relationship between primitive objects is called a basic relationship.

Distance is represented as distance \((V_1,V_2,L)\) while the angle between two lines is represented as angle \((V_1,V_2,V_3,A)\) or angle \((V_1,V_2)\), edge \((V_2,V_3)\), \(A\) = angle \((E_1,E_2)\), where \(A\) denotes the angle between \((V_1,V_2)\) and \((V_2,V_3)\).

Crossing occurs when two edges or vertices meet. The cases of crossing are edge\_cross \((E_1,E_2)\), where two edges meet; vertex\_cross \((V_1,V_2)\), where two vertices meet; and ve\_cross \((V_1,E_1)\), where an edge and a vertex meet. Vertex crossing is the same as vertex touching and an edge and a vertex crossing is the same as two vertices touching.
Parallel edges are represented as parallel \((E_1, E_2)\) and perpendicular edges are represented by perpendicular \((E_1, E_2)\).

Also, there are descriptions for the relative position of primitive objects. When \(O_1\) denotes objects then above \((O_1, O_2)\) is when \(O_1\) is above \(O_2\) and is the same as below \((O_2, O_1)\). Right \((O_1, O_2)\) represents \(O_1\) being to the right of \(O_2\) and is equivalent to left \((O_2, O_1)\). Inside \((O_1, O_2)\) represents \(O_1\) being inside \(O_2\) and is equivalent to outside \((O_2, O_1)\).

Descriptions of touching objects are inside_touch \((O_1, O_2)\), where \(O_1\) is touching \(O_2\) in the inside, and which is equivalent to outside_touch \((O_2, O_1)\). Touching can further be divided into edge_touch \((O_1, O_2)\) where \(O_1\) and \(O_2\) share a common edge; vertex_touch \((O_1, O_2)\) where \(O_1\) and \(O_2\) share a common vertex; ve_touch \((O_1, O_2)\) where a vertex of \(O_1\) and an edge of \(O_2\) touch; and partial_touch \((O_1, O_2)\) where \(O_1\) and \(O_2\) share a portion of an edge.

Multiple edge crossings are represented by overlaps. Open_overlap \((O_1, O_2)\) is when \(O_1\) and \(O_2\) completely overlap each other while hidden_overlap \((O_1, O_2)\) is when a portion of \(O_1\) completely covers \(O_2\).

![Figure 3-1 Edge crossing of edge1 \((V_1, V_2)\) and edge2 \((V_3, V_4)\).](image-url)
Figure 3-2 Vertex crossing of vertex $V_1$ vertex $V_5$.

Figure 3-3 Inside edge touch

Figure 3-4 Inside edge partial touch
3.4.3 Composition Operations

There are two types of composition operations. One is the join operation where two objects are joined, and the other is the placement operation where an object is placed at an arbitrary distance from another object.

The join operation depends on whether the objects are arranged in a horizontal or vertical direction, and on whether the object is placed on the left, right of, above, or below the other object. Examples of the join operation are:

- horizontal_join (O₁,O₂,Right)
- horizontal_join (O₁,O₂,Left)
- vertical_join (O₁,O₂,Above)
- vertical_join (O₁,O₂,Below)

The placement operation depends on vertical or horizontal placement, the angle, and whether the item is placed inside or outside another item. Examples of the placement operation are:

- place_horizontal (O₁,O₂,Right,D)
- place_horizontal (O₁,O₂,Left,D)
- place_vertical (O₁,O₂,Above,D)
- place_vertical (O₁,O₂,Below,D)
- place_angle (O₁,O₂,Angle,D)
- place_inside (O₁,O₂,Inside)
- place_inside (O₁,O₂,Outside)
Figure 3-5 Vertical_join (O₁, O₂, Above) and horizontal_join (O₁, O₂, Right).

Figure 3-6 Place_inside (O₁, O₂, Outside) and place_vertical (O₁, O₂, Right, D).
Chapter 4

Constraints

4.1 Overview

Constraints reflect a way that humans like to think about situations. In mathematics, the hypotheses of theorems, along with the fundamental axioms, may be viewed as constraints. In design, constraints represent the requirements that the design must satisfy.

Constraints are useful tools to represent the relations that must be maintained between objects. They can be used to specify relations between objects, to maintain consistency between user program data structures and graphical depiction of objects, and to specify layout. A constraint can be represented by both a declarative description of the relation and a set of procedures for making that relation hold.

To express a relation as a constraint, the following information is needed: a rule, and one or more methods for satisfying the constraint.

4.2 Literature Review

In 1969 Friedman and Leondes [Fried 69] presented the theory and application of developing an analytic foundation for constraint theory, and in
1976 Friedman [Fried 76] presented another overview for this theory. In 1983 Richard [Rich 83] presented the theory and its application. In 1990 Borning [Born 90] presented the "DeltaBlue algorithm" as an incremental constraint hierarchy solver. It maintained an evolving solution to the constraint hierarchy as constraints were added and removed. DeltaBlue minimized the cost of finding new a solution after each change by exploiting its knowledge of the last solution.

For the application in computer system development, Sketchpad [Suth 63], ThingLab [Born 79], IDEAL [Van 81], Coral [Szek 88], OTP [Arba 89], etc. had been applied.

The Sketchpad system was the first to be used as a computer system that could solve constraints. Constraints were used to specify conditions that the picture had to satisfy. For example, one could constrain a line to be horizontal, or a point to lie on a line. The domain of constraints was objects.

ThingLab extended Sketchpad in a number of ways to use constraints. To support constraints, some new kinds of objects were implemented. Constraints were also applied to non-numeric objects such as text, and which led to the development of developed constraint-based language.

IDEAL used equations as constraints, and a set of constraints was used to define the relative locations of points of the box which called the fundamental objects of which figures are composed.

Coral is a user interface toolkit based on graphical objects and constraints. The values of attributes of graphical objects are defined in term of the values of attributes of other graphical objects using constraints.
OTP was based on the operational interpretation of constraints, to satisfy a network of geometric constraints incrementally. OPT used a high-level understanding of the semantics of constraints and the geometric implications of operations for satisfaction planning.

4.3 Description of Constraints

Efficient constraint description methods as well as efficient solving methods are needed to draw two-dimensional objects efficiently. In this thesis, constraints are divided into basic constraints, requirement constraints, and default constraints.

Basic constraints are the constraints which are checked and maintained in the system. Examples are:

parallel (line1, line2)
perpendicular (line1, line2)
line is horizontal
line is vertical

Required constraints are the constraints needed to display a certain type of object. These constraints may be given by the user or set by default depending on the data type. For example, in order to draw a binary tree with one root node (a) and two child nodes (b, c), the following constraints are needed to preserve the properties of the binary tree:

connect (a, b) and connect (a, c)
arrange b and c on one horizontal line
arrange a so that it is above and between b and c.

Default constraints are constraints which are decided by the computer when there are no required constraints. The following are example cases of default constraints.

1. None of the figures displayed may overlap.
2. There must exist a consistency among identical variables or contents in terms of their size, appearance, color, thickness, etc.
3. All the figures drawn are minimized to the greatest extent possible.
4. The placement of the descriptions such as values or names must be consistently placed between the various figure drawings.
5. When figures can be depicted in more than one way, the most conventional method will be implemented.

4.4 Constraints Satisfaction

Constraint satisfaction problem is not trivial. A basic problem is that constraints are multi-directional. For example, the vertical line constraints is allowed to change either endpoint of the line. Another problem is that one constraint may interfere with another.

The Sketchpad system used two methods for constraint satisfaction. One was propagation of degrees of freedom and the other one was relaxation, which displayed the figure with errors.
The ThingLab system added one method which is propagation of known states. But the above two systems have one problem, which is that the objects will invoke a method for dealing with circularity. And the ThingLab system used the Smalltalk facilities.

CSP [Hent 89] proposed a tree search method such as backtracking, forward checking, and looking ahead as a constraint satisfaction.

4.5 Approach to Constraint Solving

One approach to solving constraints is analysis of constraints and planning how to satisfy them.

In my thesis, two methods of solving constraints are introduced. The first is the compute-and-placement method. In this method, the object placement is checked to see if it satisfies all constraints as the position is calculated and then placed. The second is the placement-and-compute method. In this method the objects are randomly placed and then checked to see if all constraints are satisfied. If the constraints are not satisfied, the objects are incrementally adjusted until they are. For instance, in Fig. 4.1 shows the display of a root node a and a son node b.
First, node a's position is represented by a circle with coordinates $(x, y)$ and a radius of $r$. Then the relative position of $b$ is calculated. Let $a (x, y, r)$ such that $x$, and $y$ are the location of $a$ and $r$ is the extent of $a$. Let $b (x + \Delta x, y + \Delta y, r + \Delta r)$ where $\Delta x$, $\Delta y$, and $\Delta r$ are initial choices which must satisfy the constraints such that:

$\Delta x = 0, 1, 2, ...$

$\Delta y = 0, 1, 2, ...$

$\Delta r = 0, 1, 2, ...$

By the compute-and-placement method

Phase I : Compute the new location using constraints

Phase II : place node $b$

By the placement-and-compute method

Phase I : place node $b$ randomly

Phase II : readjust incrementally using constraints and the boundary condition $(\Delta x_i - \Delta x_{i-1}) < E$, where $E$ is the boundary value.
In this case, the problems of in what order the constraints are to be applied and which search method is to be used for efficiency arise. To solve these, we can use the constraint-hierarchy given by user or the system. The search methods can be solved using the simulation.

Certain cases of constraint solving problems could be regarded as NP-complete problems [Rivi 89] and the speed of processing time is very important. My methods are simple and thus it is easy to apply compare my results with those of previous systems.

### 4.6 Example

The following examples show how the constraints are solved by the placement-and-compute method.

The equation of constraints solving can be represented by the multiplying the term of constraint and the steps of constraints solving can be performed by backtracking method. The backtracking method examine the first element of constraints and then examine the second element of constraints, if the first element of constraints is satisfied. If the first element of constraints is not satisfied, the system will not examine the next constraints.

Let $C_1...C_n$ be the set of constraints. Then the equation of the constraints solving, $\Phi(C_1...C_n)$, can be represented as follow:

$$\Phi(C_1...C_n) = \bigcap_{i=1}^{n} C_i$$
Let a, b, c be the nodes of a binary tree such that node a is a root node, and node b and node c are the child nodes of node a. Then, the following figures show the steps for forming the binary tree using the constraints. We assume that constraint $C_i$ can be examined earlier than constraint $C_j$ for $i < j$.

**Constraints of binary tree**

$C_1$. Node a, b, c are represented by a rectangle box with edge length l.

$C_2$. Connect (a, b) and connect (a, c).

$C_3$. Horizontal (b, c).

$C_4$. Right (b, c), left (b, a), right (c, a)

$C_5$. Below (b, a) and below (c, a).

$C_6$. Node a is the middle of node b and node c.

$C_7$. Unit distance (a, b) and (a, c).

$C_8$. 30 degree $<=$ angle (child_node, root, center_line) $<=$ 60.

![Diagram](image_url)

**Figure 4-2** Initial random placement with $C_1, C_2$
Case 1

Case 2

Case 3

Figure 4-3 With constraint C7

Case 1

Case 2

Case 3

Figure 4-4 With C4, C5

Case 1

Case 2

Case 3

Figure 4-5 Add node c with C2, C4, C5
Figure 4-6 With C₃, C₆

Figure 4-7 With C₇
Chapter 5

Data Extraction

5.1 Overview

Data extraction is the process of converting the user's data structures in the runtime environment into standard form such as symbol table description language, which can be used for figure drawing and display. An algorithm called a data extraction algorithm is developed for extracting the data from the program symbol table. From the symbol table description language, we can create the standard data structures of 2-D objects, and these are mapped onto the graphical description language for drawing of 2-dimensional objects on the screen. In order to map efficiently, a mapping language must be created.

A standard data structure of 2-D objects can be seen in the figure elements (FigEl). The use of figure elements, FigEls, in data structures can be seen in Akrti [Desa 88], xgdb [Garg 89], and Radack & Mateti [Rada 89]. These systems are used only for the display of data structures, and an upgraded FigEl data structure is needed to draw general two-dimensional objects.
Figure 5-1: Overview of the system

A data extraction algorithm can be seen in Radack & Mateti [Rada 89].

Most of the information about the two-dimensional objects can be found within a debugger’s symbol table. However, when this information is extracted from the debugger’s symbol table, the 2-D object display system becomes debugger dependent. In order to build a more general system, a
separate language was designed for the description of the symbol table. In this thesis this language is simply called a symbol table description language. In addition, to create FigEl from the symbol table description language, a mapping language was designed and to map information from FigEl on to the graphical description language, another mapping language was developed.

5.2 Data Structure Representation

5.2.1 Description of FigEl

Figure elements (FigEl) are defined as primitives or a group of primitives connected with constraints. Once the data structures for the figure element are decided upon, and the two-dimensional objects are redefined as figure elements, then the object can easily be drawn on the screen.

On the screen, each basic building block of the figures is called a primitive object. These primitive objects can draw simple figures. In order to draw more complex figures or data structures, these primitive objects are used in a composite through special placement operations such as "join," "placement," etc. As the typical examples of the composing operations of FigEl, juxtaposing, connecting, and collecting [Rada 89] can be seen. Juxtaposed elements are arranged in a line so that they touch. For instance, structured types of variables: array, records, and such, are represented as juxtaposed composites, where each element is represented with one figure element, and the whole is represented as a composite of figure elements. In a
connection operation, the elements are linked with an arrow. For instance, when numerous records are formed as one figure, the connection operation is used to connect each record. A collection operation is done when several smaller figures are present within a larger figure. Here, the larger figure, and the enclosed figure do not have any relationships.

5.2.2 Data Structure of FigEl

FigEl data structures must be designed such that any two-dimensional objects can be drawn. Also, all the above operations must be made possible. To fulfill these criteria, the following data structure is needed.

1) A FigEl consists of an object’s name and type, the object’s location and the object’s figure description parts. The FigEl data structure describes one figure element of one graphical picture. When a graphical picture consists of more than one figure element, the relationships are contained within the data structure.

2) The object’s name is represented by a character string. The object type is represented as a variable type and as an abstract type. Variable types are primitive object types, such as integer, real, float, etc. and structured types such as array, pointer, and record. Abstract types are represented with the same data structure, but can represent 2-D objects such as polygons, rectangles, binary trees, graphs, linked lists, etc. The
abstract type is drawn on the screen by default unless hinted at by the user.

3) Objects are also represented by id-location and location. Id-location represents the object’s relative position at which it is drawn, and location represents the location of the FigEl in its parent coordinate system.

4) The object’s figure description represents whether the object is of an atomic type, where the object consists of one figure element; whether several figure elements were composited; whether the figure elements are connected by arrows, as in trees, linked lists, and graphs; or whether figure elements must be drawn multiple times. The description must also represent their subfigures as in case of trees and linked lists, left siblings, right siblings, and parents, as well as represent subfigures and targets for pointers, and have a sequence for out pointers. Also, an area must exist for insertion of values.

**Data Structure of FigEl**

\[ \text{figel}(\text{Label, Name, atomic, Type, Value}) \]

: for integers, floats, enumerates, characters, others.

\[ \text{figel}(\text{Label, Name, juxtaposed, Type, [ Label}_1 | \text{ Label}_2 |...| \text{ Label}_n ]) \]

: for arrays, single structures.
figel(Label, Name, network, Type, ([ Label₁ | Label₂ | ... | Labelₙ ],
[ Pointer₁[ Label₁, Label₂ ] | Pointer₂[ Label₂, Label₃ ]
|...| Pointerₙ[ Labelₙ₋₁, Labelₙ ]))).

: for structures with pointers.

figel(_, _, disconnected, _, [ Label₁ | Label₂ |...| Labelₙ ]).

: where labels are atomic, juxtaposed, or network.

5.3 Symbol Table Description Language

5.3.1 Overview

When the information needed for the drawing of two-dimensional objects onto the screen is extracted from the debugger's symbol table, the display system becomes dependent upon the debugger. To avoid this debugger dependency, a separate symbol table description language was designed. Thus, as long as the same symbol table description language is used, the debugger will not be a variable in the performance of the display system. If we change the data extraction algorithms, the 2-D objects display system can be easily implemented on any computer system.
5.3.2 Syntax of the Symbol Table Description Language

We will now describe the syntax of the symbol table description language (SDL). SDL reserved words are represented by capital letters.

1. The SDL can express all types of variables. The SDL syntax starts with "Start" and ends with "End."

   Start
   variable-expression.
   .
   .
   End.

2. Variable-expression is expressed as a variable name with an address.
   
   "variable" is an SDL reserved word.
   variable ( Variable-name, Address ).

3. Value-expression is represented by variable types and values. Nested value-expressions are possible for array variables and nested variables are possible for record types. "[" and "]" depict a nested structure.
   "value" is an SDL reserved word.

   value ( Address, Value-types( Values ).
   value ( Address, Value-types([ Value-expression]).
4. Variable names are represented as only an identifier, a "*" plus an identifier. "*" plus identifier represent a pointer variable.

   Variable-name:
   identifier
   *identifier

5. Variable types are integer, real, long, float, pointer, string, array, and record.

   Variable-types:
   integer
   real
   etc.

6. Values are numbers or identifiers and when values is a pointer variable case, values is represented by address.

   Values:
   numbers
   +identifier
   Address

7. Identifiers are a sequence of letters and digits starting with letters. Letters are "A" thru "Z" and "a" thru "z," and digits are "0" thru "9."

8. Address is represented by "v" plus unsigned integers.
Address;
v+unsigned integers.

9. Numbers are all represented as floating point.

\[
\text{number::} \\
\text{ +/- unsigned-integer. \{unsigned-integer\}} \\
\text{ E +/- unsigned-integer.}
\]

10. Unsigned integers are sequences of digits.

### 5.3.3 Typical Examples

#### 5.3.3.1 Single Variable

For the following C code,

```c
char name = 'sklee';
integer isum = 100;
real sum = 10.5;
float diameter = 0.34E-3;
long k = 3;
integer *p;
*p -> integer(50).
```

The SDL expression would be

- variable(name, v00001).
- variable(isum, v00002).
- variable(sum, v00003).
variable(diameter, v00004).
variable(k, v00005).
variable(p, v00006).

value(v00001, char('klee')).
value(v00002, integer(100)).
value(v00003, real(0.105E2)).
value(v00004, float(0.32E-3)).
value(v00005, long(3)).
value(v00006, pointer(v00007)).
value(v00007, integer(50)).

5.3.3.2 1-Dimensional Array

For the following C code,

```c
Integer stack[2];
    stack[0] = 11;
    stack[1] = 22;
    stack[2] = 33;
```

```c
float queue[2]:
    queue[0] = 23.5;
    queue[1] = 34.6;
    queue[2] = 45.7;
```

The SDL expression would be

```c
variable(stack, v00001).
variable(queue, v00002).

value(v00001, array(3,[v00003, v00004, v00005])).
value(v00002, array(3,[v00006, v00007, v00007]),
```
v00008}

value(v00005, integer (11)).
value(v00004, integer (22)).
value(v00005, integer (33)).
value(v00006, float (23.5)).
value(v00006, float (34.6)).
value(v00006, float (45.7)).

5.3.3.3 2-Dimensional Array

For the following C code.

```c
integer matrix[1, 1];
matrix[0, 0] = 5;
matrix[1, 0] = 6;
matrix[0, 1] = 7;
matrix[1, 1] = 8;
```

The SDL expression would be

```
variable(matrix, v00001).
```

```
value(v00001. array(4,[v00002, v00003, 
v00003, v00004])).
value(v00002. integer(5)).
value(v00003. integer(6)).
value(v00004. integer(7)).
value(v00005. integer(8)).
```

5.3.3.4 Structures/Records

For the following C code,

```
structure node
```
{  
    char info;
    integer lson;
    integer rson;
} [2];  

The SDL expression would be

variable( node, v00011).
value(v00011, structure([[(info, v00012),
                          (lson, v00013),
                          (rson, v00014)],
                         [(info, v00015),
                          (lson, v00016),
                          (rson, v00017)]].)

value(v00012, char()).
value(v00013, integer()).
value(v00014, integer()).
value(v00015, char()).
value(v00016, integer()).
value(v00017, integer()).
value(v00012, char()).

5.4 Extraction Algorithm

Because a picture may consist of one component, or any number of components and at times many disconnected components, a question arises on how to best divide the variables and display the variables on the screen. The Radack and Mateti system [Rada 89] solved this problem by first finding a "weakly connected component" by using a depth-first search algorithm.
Through the use of the symbol table description language, an easier method of extraction could be realized. Since the SDL syntax keeps the weakly connected components separately, figure elements can be created while traversing the SDL, i.e. checking for the reserved words variable and value.

SDL contains complete information on each variable. Therefore, the building of the extraction algorithm is simplified. The following is Radack & Mateti's extraction algorithm.

**Algorithm Map_to(SDL)**

The input is a set of variable references. The output is an SDL representation for two-dimensional objects.

1. Find all weakly connected components. Call them $C_1$...$C_n$.

2. For each weakly connected component, find the set of source variables. Let $V_i$ be the set of source variables for component $C_i$.

3. For each $i$ between 1 and $n$, Map_Sym_to(SDL).

4. Join the object $O_i$ ($1 \leq i \leq n$) into the disconnected figure.

**Algorithm Map_Sym_to(SDL)**

The input variable is a set of variable references $V$ which are known to be in a weakly connected component. The output is a SDL representing a connected (network) figure element.

1. Initialize the queue to empty.
2. Invoke Map_Val_to(SDL on each variable reference in V. Save the resulting SDLs.
3. While the queue is not empty, do steps 4 to 7.
4. Remove (SDLs, Val) from the queue, where SDLs is a SDL representation of a FigEl and Val is a value.
5. Convert Val into pointer addr using a built-in function and check the figure table to see whether there is an entry for it.
6. If it was not found, then invoke Map_Val_to(SDL on fig and save the resulting SDL for. Store (addr, ft) in the figure table.
7. Set the target field of fig to point to ft.
8. Join together all the SDLs created in step 2 and 6 as children of a network SDL.

Algorithm Map_Val_to(SDL

The input is a value. The output is a connected (atomic or juxtaposed) SDL.

1. Look up the type of the input value.
2. If it is an atomic type (integer, real, set, etc.), convert its value to a string and create an atomic SDL to hold it.
3. If it is a composite type (record or array), obtain the values for all the components and recursively invoke Map_Val_to(SDL to convert them to SDLs. Make them children of a juxtaposed SDL.
4. If it is a pointer type, create an atomic SDL and center it and the pointer value (an address) into the queue.
5. Return the SDL created in step 2, 3, or 4.

Algorithm SDL_to_FigEl

The input is SDL and the output is FigEl.

1. Take one SDL statement.
2. Add the FigEl type.
3. Map to FigEl for each component of SDL using the mapping language.
Chapter 6

Layout Algorithms

6.1 Overview

Section 5 dealt with the creation of FigEls. This section deals with how to display those elements on the screen.

When a figure is drawn on a screen, the following criteria must be kept in mind. First, the size of the displayed figure must be decided. Though larger figures are easier to read, the amount of information that may be displayed decreases as the figures grow larger. DDS [Ojed 85] displayed the figures in an original size, while Aktis [Desa 88] used pan and zoom in the display while the size was user decided. xgdb [Garg 89] and Radack & Mateti's system [Rada 89] were designed so the displayed figures would be the smallest size that could contain the variable names or values. The optimal method would be a system that would use the smallest possible figure in which labels could be inserted, while also having the ability to pan and zoom.

Second, the distance between two figures (when more than one figure is drawn on the screen) must be determined. Here is where network FigEls and disconnected FigEls appear. Too short of a distance between figures will cause difficulty in reading, while large distances will need proportionally more display room. If the screen can be scrolled, this becomes less of a problem. DDS [Ojed 85], Aktis [Desa 88], xgdb [Garg 89], Radack & Mateti [Rada 89].
as typical of most systems, use minimum separation. Minimum separation in
the drawing of data structures is well explained in Ding and Mateti [Ding 89].

Finally, routing must be determined. The most common method is
preserving the aesthetic qualities of each figure, while minimizing edge
crossing.

6.2 General Layout Algorithms

A general algorithm is an algorithm which can layout a component of a
two-dimensional object to be drawn-simple objects can be represented by
FigEls; complex objects are represented by more than one FigEl. These are
displayed using atomic FigEls and composites FigEls.

6.2.1 Atomic FigEls

Atomic FigEls are the most basic figure element representing simple
objects, such as a single component of a complex object. In the case of data
structures, scalars, integers, reals, booleans, a single component of an array or
of a record, a single node component of a graph, tree, or linked list can be
represented by an atomic FigEl. The atomic FigEls represent as a minimum
certain primitive objects which can enclose the information for the element.
6.2.2 Composed FigEls

Complex objects are represented by composite FigEls, and are drawn by compositied operations such as "join" or "place" operations.

The most common examples of composite FigEls are arrays. The layout for arrays include whether the components are lined up vertically or horizontally, and how to draw the box that will bound the array [Desa 88, Rada 89]. In the case of arrays, where all the figure elements must be the same size, the largest component size is chosen as the size for all the components.

The array drawing can be accomplished by horizontal join operations or vertical join operations. The array layout also include a variable which can indicate direction, and the ability to calculate a box which will bound the array subparts with the minimum dimensions. One-dimensional, and two-dimensional arrays need only one bounding box, while three-dimensional arrays will need a bounding box around the bounding boxes.
6.2.3 Disconnected FigEl's

The size of the screen area must be taken into account when displaying multiple disconnected FigEl's. A simple algorithm which will squeeze the maximum number of figures onto the screen will not suffice due to aesthetic reasons. In this thesis, a minimum distance is given, and a greedy method algorithm is used. Here a memory allocation plan such as the Best-fit and First-fit Algorithm [Pete 82] can be considered. The following is a placement algorithm using the greedy method [Rada 89].

1. From the set of unplaced items, select the item with the largest area.

2. Place this item so that it does not touch any already-placed items, and so that the area of the bounding rectangle of all the placed items (including the current on being placed) is minimized.
3. If there are remaining items, then go to step 1.

(A)

<table>
<thead>
<tr>
<th>New element E1</th>
<th>blank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Already placed elements A1</td>
<td>A3</td>
</tr>
<tr>
<td></td>
<td>A2</td>
</tr>
</tbody>
</table>

(B)

<table>
<thead>
<tr>
<th>blank</th>
<th>New element E1</th>
<th>blank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Already placed elements A1</td>
<td>A3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-3 Placement A is preferred over B because it leaves a large area within the bounding rectangle (blank) for placement of other figures [Rada 89]

6.3 Special Layout Algorithms

A special algorithm is needed for the display of network FigEl's such as trees, graphs, and linked lists. Each node becomes a FigEl, and edges or arrows must be routed between the FigEls. In the course of programming, trees, graphs, and linked lists are usually programmed using arrays and
records, and at times with pointers. Therefore, trees, graphs, and linked lists are not at all obvious. So when displaying these types, the user must give the abstract type, and when no abstract type is given the data is displayed as a linked list. This section deals with the layout for linked lists, binary trees, m-ary trees, and graphs.

6.3.1 Linked Lists

Linked lists can be divided into single linked lists, double linked lists, and circular linked lists. The node placement on the screen is the same for all cases, and only the link routing differs. Linked lists are laid out in either a horizontal or vertical direction, and each following node's placement must be decided before display, and placed at a distance with minimal separation to the next node. Algorithms for the position of the arrow for routing can be seen in Desai [Desa 88], and Garg [Garg 89].

![Figure 6-4 Linked list](image-url)
6.3.2 Binary Trees

Binary tree layout algorithms can be seen in papers such as Wetherell & Shannon [Whth 79] and Reingold & Tilford [Rein 81]. In Wetherell's algorithm, a count holder was kept at each level and a one unit gap was maintained for the height and the node's x-coordinate. This algorithm uses a top-down method, where the father and left son are always on a vertical line. An implementation of Reingold's algorithm can be seen in Radack & Mateti [Rada 89].

The Reingold and Tilford's algorithm operates on a binary tree data structure. Along with the normal child pointers, each node contains fields for: (1) the horizontal offset of the children from the current node; (2) a bit indicating whether each child pointer is a thread; and (3) the "information" content of the node. In any case, the information is a pointer to the FigEl representing the picture to be drawn for that node.

If the debugger's subsystem were used, a binary tree could not be displayed until the program was finished running.

Figure 6-5 A binary tree, by the default layout algorithm
6.3.3 M-ary Trees

Algorithms concerning m-ary trees can be seen in Wetherell & Shannon [Weth 79], Radack [Rada 88]. Wetherell's algorithm is basically the same as the binary tree algorithm. Lee's algorithm does minimize drawing space, but the algorithm is not aesthetically satisfactory. Radack's algorithm, improved upon Reingold's algorithm, is both more aesthetically pleasing and faster than the other algorithms.
6.3.4 Graphs

Graph layout algorithms can be said to never be perfect, because of the innumerable types of graphs. Therefore, a graph algorithm can be considered to be malleable for each purpose. A generalized algorithm which includes trees can be seen in Edge [Newb 89]. Tamassia’s algorithm [Tama 61], Belhandouz’s theses [Belh 90], xgd [Garg 89] show algorithms which concentrate on readability. The next section introduces a new algorithm which is based on minimizing the average length of the edges.

6.3.5 Routing Algorithms

Routing is the drawing of arrows between the displayed figure components. The objective of routing algorithms is to minimize the length of the arrows; to minimize edge crossing; and to maximize aesthetics. In this thesis a simple routing algorithm for layout is used.

Routing Algorithm

1. Draw a line from the center of the figure component representing the pointer to the center of the nearest edge of the pointee.
2. If step 1 is not possible, do step 3 to 4.
3. Draw a line starting from the center of the pointer figure component going up or down, or left or right with unit distance from the previously drawn arrow minimizing edge crossing.
4. Continue drawing horizontally or vertically until the center of the pointee is reached, then move perpendicular until touching the nearest edge of the pointee figure component.

6.4 Graph Layout Algorithm

We compare the performance of certain graph layout algorithms. The algorithms are grid-based; i.e., all nodes are placed on a rectangular grid, and they are designed to minimize the average length of connections between nodes.

6.4.1 Introduction

The goal of a graph layout algorithm is to map the graph onto the plane in such a way as to satisfy certain constraints and to optimize certain properties, both of which depend on the application. Graph layout is important in visualization of graphs and for design automation of VLSI.

In both applications, nodes may be represented as rectangles. We assume that the rectangles are of uniform size - \( l_x \) by \( l_y \), and must be separated by a minimum separation distance \( m_x \) and \( m_y \) in \( x \) and \( y \) respectively. Thus, the center of the nodes must be separated by

\[
d_x = l_x + m_x
\]

in \( x \) and

\[
d_y = l_y + m_y
\]
in y.

The class of algorithms we are considering place the nodes on a rectangular grid. Thus, the center of each node will be placed at

\[ [i, j] = (i'd_x, j'd_y) \]

for some integer value of i and j.

Let \( G = (V,E) \) be a graph. We will denote the position of vertex \( v \in V \) as \( P(v) \).

A layout algorithm must also route edges, generally so as not to cross nodes. We will assume, however, that edges are straight lines connecting the nodes.

Among the properties which may be optimized by a layout algorithm are: (1) the number of edge crossings; (2) the maximum number of edge crossings at any particular point; (3) the area of the bounding rectangle of the placed nodes; and (4) the "average" or maximum length of an edge.

We choose to minimize the "total distance between connected edges"

\[
\sum_{(v,w) \in E} d(P(v),P(w))
\]

which is related to the average length of an edge.

Unfortunately, this problem has been shown to be NP-complete by polynomial reduction to the quadratic assignment problem [Kurt 62].

Therefore, we use heuristics.
6.4.2 Previous works

Breuer addresses graph layout from the viewpoint of application to circuit layout [Breu 72]. They have grouped the placement techniques into the three classes: (1) constructive initial-placement; (2) iterative placement-improvement; (3) branch-and-bound. In the main, the first and third categories are constructive; in the second, they are interactive. The iterative algorithms seek to improve a placement by repeated modification of it. At every stage there is a complete placement available. Thus, an iterative algorithm may be interrupted at end of any cycle. The constructive algorithms, on the other hand, produce a placement configuration only upon termination.

In constructive initial-placement methods the placement configuration is formed by adjoining modules to a subset of already placed modules. This class is illustrated with algorithms due to the pair-linking method and cluster-development method [Kurtz 65]. In those algorithms, the tie-breaking rule (one step look-ahead) is used for solving not to a unique module.

In general, iterative placement-improvement methods demand more compute time than the constructive initial-placement methods but are potentially capable of producing better layouts. These methods are shown in the Steinberg assignment method [Stei 61], relaxation method [Fisk 67], pairwise interchange method, stochastic method, etc.

Branch-and-bound methods are applicable to the quadrartic assignment problem [Gilm 62] and can be used to find an optimal solution to that problem if the order is less than fifteen modules. The branch-and-bound method proceeds as follows. The set of all feasible solutions is partitioned,
and a search for the optimum is made in each partition. A lower bound is computed for the solutions in each partition, and the search in any partition is terminated upon the lower bound exceeding the cost of some previously feasible solution.

In this paper, the graph layout algorithm is a type of constructive initial-placement method. But the solution of the tie-breaking case is the branch-and-bound method. That is, the N-lookahead method is used for solving of the tie-breaking problem.

6.4.3 Graph Layout Algorithm

6.4.3.1 Overview

This graph layout algorithm consists of two parts: the first is the selection of a node to be placed, and the second is choosing a placement for the node.

The methods of the node selection are based on the number of edges of each node. The basic idea is that the node with the most edges should be selected first. This idea leads to more efficiency for minimizing the total distance. Two method are explored. The first method of node selection is related to the most total edges and the second one is related to the most edges with placed node already.

The method of node filling is that the first node is filled at the middle of the grid. and that the next node is filled around the first filled node, so as to
minimize the total distance between the nodes placed so far. The lookahead that is used in this method gives better results. But the lookahead method needs more time complexity. For reducing this time complexity, we can use the enclosing ring method.

The enclosing ring is the set of blank cells which surround the non-blank cells. The next P nodes need to be placed at one of the surround cells of nodes which have been placed already so as to minimize the total distance between connected edges. In the enclosing method, the selection of grid-cells for filling is carried out only in the enclosing ring, instead of all the grid-cells. Hence, this method can give a reduction of grid-cell selection time.

6.4.3.2 Terminology

In this paper, we use some terms to describe the algorithm as follow: ENCLOSING_RING is the set of blank cells which surround the non-blank cells. GRID-CELL is the set of cells for filling the nodes. V is the set of nodes in the graph. Vp is the set of nodes which have been placed already. SELECTED-NODE-QUEUE (which holds nodes that are the next candidates for placement) is a data structure.
6.4.3.3 Graph Layout Algorithm

6.4.3.3.1 Input

The input to the algorithm is an adjacency matrix \( M = m(i,j) \),

where

\[
m(i,j) = \begin{cases} 
1, & \text{if there is an edge from } V_i \text{ to } V_j \\
0 & \text{otherwise.} 
\end{cases}
\]

This adjacency matrix and the node names are used as input data.

6.4.3.3.2 Grid and Enclosing Ring

In this paper, we use a rectangular grid, and the first grid is created with the only one cell at middle of grid \( G[0,0] \). Place the first node at \( G[0,0] \).

The enclosing ring is used for reducing search time of grid-cell selection. The first enclosing ring is the set of blank cells which are the nearest surround cells of the non-blank cells. The second enclosing ring is the set of blank cells which are the nearest surrounding cells of the first enclosing ring.

RECREATE ENCLOSING RING: [1] Check first enclosing ring of the last-filled location: if there exist blank cells that are not members of the first enclosing ring, then add those cells to the first enclosing ring and delete
those cells from the second enclosing ring. [2] Check the second enclosing ring of the last-filled location: if there exist blank cells that are not members of the first and second enclosing rings then add to the second enclosing ring.

This method reduces the selection time of blank cells for node filling.

The fig 6.7 shows the non blank cells with letters, the first enclosing ring (light gray) and the second enclosing ring (dark gray).

![Figure 6-7 Rectangular grid](image)

**6.4.3.3.3 Algorithm**

select the first node from node-set with constraints and place it in selected-node-queue.

while remain node-set is not empty
{
    select the next node with constraints and place it in selected-node-queue.
}
create the first grid cell
select the first node from the selected-node-queue
and placed it at G[0,0] in grid-cell.

create enclosing-ring with G[0,0].

while the selected-node-queue is not empty
{
    choose the next node from selected-node-queue
    place it at G[x,y] in grid-cell with lookahead
    such that

    \[
        \text{MIN } - \text{dist}(N_i, N_j)
    \]
    exists an edge
    between N_i and N_j
    where N_j is the member of placed-node
    recreate enclosing-ring.
}

print out the results.

6.4.3.4 Node Selection Method

We select a node for filling with some constraints so as to minimize the
total distance between connected nodes. In this paper, we will give two node
selection methods. Method 1 is based on already placed nodes and Method 2
is based on the most total edges.
6.4.3.4.1. Method 1

FIRST NODE SELECTION: Select the node with the most edges. If there exist two or more nodes which have equal number of edges, then we can select one node arbitrarily.

NEXT NODE SELECTION: [1] Select the node that has the most edges to already placed nodes. If there exists one then exit this routine, otherwise goto step 2. [2] Of the nodes from step 1, select the node with the most total edges. If there exists one then exit this routine, otherwise goto step 3. [3] Of the nodes from step 2, select the node with an edge to the earliest placed-node. If there exists one then exit this routine, otherwise goto step 4. [4] Of the nodes from step 3, select one node arbitrarily.

6.4.3.4.2 Method 2

FIRST NODE SELECTION: Select the node with most edges. If there exist two or more nodes which have equal number of edges then we can select one node arbitrarily.

NEXT NODE SELECTION: [1] Select the first pivoting node from the earliest placed node. (The pivoting node is the earliest placed node. When the node selection is performed, we check the adjacent nodes of the pivoting node first.) [2] Select the node that has the most edges and that has an edge to the pivoting node. [3] From step 2, if there do not exist any nodes and the
remaining node set is not empty, then select the new pivoting node from the next earliest node and goto step 2. [3] Of the nodes from step 2, if there exists one then exit this routine, otherwise goto step 4. [4] Of the nodes from step 3, select one node arbitrarily.

6.4.3.5 Node Filling Algorithm with n-lookahead

\[
\text{Selectmin}(\text{Node,Placed-nodes,Remaining-nodes,Lookaheads})
\]
\[
\{
\text{minvalue = _;}
\]
\[
\text{For each cell in surround do}
\]
\[
\{
\text{Add node to Placed-nodes;}
\text{Remove from Remaining-node;}
\text{Compute Sum1 of dist}(N_i,N_j)
\text{where exists an edge between } N_i \text{ and } N_j
\text{and } N_j \text{ is the member of Placed-nodes:}
\]
\[
\text{if } ( \text{Lookahead} \geq 1 ) \text{ then}
\text{sum2 = Selectmin(Node,Placed-nodes, Remaining-nodes,Lookahead-1);}
\]
\[
\text{Sum = Sum1 + Sum2;}
\]
\[
\text{if Sum < minvalue then}
\text{minvalue = Sum;}
\]
\[
\}
\text{return Sum;}
\]
6.4.4 Analysis

Let $N$ be the number of nodes and $M$ be the number of lookaheads. The complexity of selection methods is a $\log_2 N$ in the average case and $N^2$ in the worst case. The node filling algorithm with $M$-lookahead has a $N^M$ complexity.

6.4.5 Example

![Graph Diagram]

Figure 6-8. Sample graph

The following example the pictures of sample graphs used by random selection and by the graph layout algorithm.

Example) Let $M$ be the adjacency matrix of graph $G$ with $n = 8$ and the rectangular grid be the pictures of results.
Figure 6-9. Adjacency matrix of G

\[
M = \begin{bmatrix}
0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0
\end{bmatrix}
\]

total distance = 25.40

Figure 6-10  By Random selection
The above example shows that the total distance by the graph layout algorithm is shorter than by random selection.

### 6.4.6 Analysis of Results

![Graph showing average distance vs. number of lookaheads]
6.4.7 Conclusion

The graph layout algorithm shows that the more lookaheads used the shorter the distance. However, this algorithm requires more processing time. This algorithm can be applied to display the record of the vertices of the graph into the screen with minimized edge crossing.

If we use this algorithm with 3 or more levels of lookahead, it is too expensive. For application problem solving, therefore, the 2-lookahead is enough.
Chapter 7

Implementation

7.1 Overview

This thesis discusses a system for the automatic layout and display of 2-D objects. The main goal is to be able to display data concerning the executing programs without any modifications to the source program. Gdb, currently the most widely used [Stal 88] on the 4.3 BSD Unix operating system, was implemented. Gdb is a fairly sturdy debugger from the Free Software Foundation which is growing in popularity in the activities of creating software design environments.

Gdb is used during program execution for the purposes of error debugging. While Gdb is in use, information on every object and variable in the executing program are recorded in Gdb's symbol table. This information can be extracted from the symbol table and mapped onto the Symbol Table Description Language for display. This is called the data extraction procedure and is handled by a data extraction algorithm. Next comes the FigEl data structure procedure, where the object information recorded in the SDL format is used to construct data structures of the object for display onto a screen. The Prolog programming language was used for FigEl construction.
Figure 7-1 Overview of Implementation

One of the main advantages of using Prolog is that the language is similar to the SDL format and thus the input data can be taken in directly without the use of parsing, thus leading to a significant reduction in processing time. Also, Prolog has constraint solving features built in, thus having the advantage over other languages in constraint solving. Next, the constructed FigEl’s are used to draw the objects onto a screen. For this purpose, a Graphical Description
Language (GDL) must be implemented. The Prolog programming language does not have functions for graphical drawing, and so the C programming language was used. Also, for the display, an interface with the X window system was needed. The X window system is a widely used windowing system for the Unix environment. A window for the display of the objects was designed to be opened automatically by the system, and after the display of the objects using a layout algorithm, the window is closed by the user moving the mouse to quit. Fig. 7-1 shows the implementation sequence of this system.

7.2 Gdb

Debugging is a mechanism which helps the programmer detect and locate errors. Debugging techniques originated with low-level programming languages. However, there are now high-level debugging systems which work for high-level languages.

There are two symbolic debuggers which are widely used on 4.3 BSD Unix operating system: gdb and dbx. Gdb is more robust and better supported in comparison with dbx. Gdb has better features and is a continuously upgraded project of Free Software Foundation located in MIT. Gdb has a built-in mechanism which allows the user to use just enough characters in a command to make the command unique. This effort was made to minimize typing. With dbx, the entire command must be used. Gdb allows multiple macros which means that the user can define a set of commands to execute multiple commands, while dbx allows only single
command macros. Gdb has much clearer code and presently supports on the
C programming language, while dbx supports three languages: Fortran,
Pascal, and C.

7.2.1 Gdb features

Gdb is run by a shell command "gdb" and the user needs to pass the
name of the executable file as a parameter. The user can start running the
program by the command "run" and cancel operation by the command "kill." There is a command "cd" to change the working directory of gdb. The input
and output can be redirected to files using > and < symbols.

The user can set breakpoints at a line, in a function, or at some specific
address in any of the files of the program with the command "break." Temporary breakpoints are set using the command "tbreak." The breakpoints
are cleared from a specific position by using the command "clear" and can also
be deleted by using the command "delete." After a break, the program can be
continued using the "cont" command. After a break, the programmer can
single step, in other words, execute the next executable line by using the "step"
command.

The source lines of the program can be listed by using the command
"list." The data can be examined by using the command "print." The user can
print the data in appropriate format by passing the format to the print
command. The command "x" can be used to examine memory under explicit
control of formats, without reference to the program's data types. If the user
wants to print the value(s) of an expression(s) each time the program stops, he can do this by executing the command "display" and passing the expression as a parameter.

The command "info sources" prints the names of all source files about which there is debugging information. The command "info functions" prints the names and the data types of all the defined functions. The command "info variables" prints the names and data types of all variables that are declared outside of functions. The command "info types" prints all data types that are defined in the program. See [Stal 88] for more information about the commands.

7.2.2 Symbol Table Structure

When the gdb is run, it first analyzes the parameters and finds the names of the executable file and the corefile, if any, then executes the functions "exec_file_command()" and "symbol_file_command()." In the first function, it reads the header structure from the executable file and assigns the values to the variables data_start, data_end, text_start, text_end, symtab_start, symtab_end, etc. In the function symbol_file_command(), it reads the symbol table from the executable file and arranges it in its own format as described below.

Gdb has a linked list of structures of type "symtab" defined in the file "symtab.h." Each of these structures is a complete symbol table for one compilation, i.e. the source file/files which are compiled together in a single
compilation. This struct contains a field "next" which links the symbol tables of different files. The field "blockvector" contains an array of pointers to a struct of type block for that file and the number of blocks in that file. Struct block contains information about all the symbols in a single block of the language, i.e., symbols defined between N_LBRAC and N_RBRAKE symbols. Each block represents one name scope. Each lexical context has its own block. The first two blocks in the blockvector are distinctive. The first one contains all the symbols defined in this compilation whose scope is the entire program linked together. The second one contains all the symbols whose scope is the entire compilation excluding other separate compilations. In C, these correspond to global symbols and static symbols.

Each block records a range of core addresses for the code that is in the scope of the block. The first two special blocks give, for the range of code, the entire range of code produced by the compilation that the symbol segment belongs to. The blocks appear in the blockvector in order of increasing starting-address, and, within that, in order of decreasing ending-address. This implies that within the body of one function the block appear in the order of a depth-first tree walk.

Block has one of its components as an array of pointers to a struct of type "symbol." This struct contains complete information about a single symbol. One of the fields of this struct "block" is "superblock" which is a pointer to the parent block, if any. These structures are defined in the file symtab.h.
After generating the symbol table it's own format, it executes the function core_file_command() if the user mentions the corefile. In this function, the different variables data_start, data_end, stack_start, stack_end, etc. are assigned the values from the information in the corefile. All these functions and the related functions are defined in the files "core.c" and "symtab.c." Then it executes the commands in the "gdbinit" file, if any, and then enters an infinite loop, reading the commands from the user and executing them.

7.2.3 Mechanism for Adding Command

As explained above, gdb has a linked list of all the commands. It has a global variable "cmdlist" which points to the beginning of the list. It has a function add_com() for adding a new function to the linked list. The syntax of the add_com command is:

```c
add_com (name, class, fun, doc)
char * name;
int class;
void (*fun)();
char *doc;
```

name is the name of the command. Gdb commands are divided into classes. class is one of those classes. fun is pointer to the function to be executed. doc is a pointer to any documentation for this command.
Every new file to be added to the source code of gdb should contain static function initialize(). This function should include one add_com () command for each new command to be added to gdb.

7.3 Data Extraction

In order to draw or print variables, the debugger must be able to give an address to each variable during program execution. For this reason, a program must be compiled and linked with the "debug" or "symbol table" option. This method differs from operating system to operating system, but the Unix operating system includes this in the executable file. In this sytem, all the needed information is extracted from gdb. The gdb symbol table contains information about the variable and type definition. In the case a symbol has a variable name, the symbol table entry contains a value entry. A value is a pointer to structure containing information about the core address, type definition, size, etc., a variable or a component of a variable.

In order to extract data from the gdb command routine, a "draw" command was added. The "draw" command displays the variable from the current frame. When a name of a variable is typed in after the draw command is given, the variable is checked to be unique, then is stored in a queue, from where one is taken at a time, and the value and variable type are taken from the gdb symbol table. Afterwards, according to the type of the variable, the symbol table description language maps the variable name, value, variable type, etc. When the variable type is a pointer, the variable that
the pointer is pointing at is looked up from the gdb's symbol table. When the variable type is an array or structure variable, a queue is built with the values and types and stored, then mapped with the symbol table description language. The above algorithms are given in section 5-4, in algorithm Map_to(SDL), algorithm Map_Sym_to(SDL), and algorithm Map_Val_to(SDL).

7.4 Symbol Table Description language

Since the implementation of the Symbol Table Description Language is almost identical to the Prolog programming language syntax, there is no need for additional modifications such as parsing. Fig. 7-2 shows creation directly from a program.

7.5 Construction of FigEl

The construction of FigEl's from the Symbol Table Description Language was in the Prolog programming language. The data structure of the FigEl is the same as the one given in section 5.5.2, and the mapping was done by the algorithm SDL_to_FigEl given in section 5.4. Fig. 7-3 shows the construction of a FigEl using Fig. 7-2 as an input.
variable(a,v00001).
variable(arr,v00002).
variable(b,v00003).
variable(c,v00004).
variable(e,v00005).
variable(pa,v00006).
variable(s,v00007).
variable(ps,v00008).
variable(t,v00009).

value(v00001,integer(7)).
value(v00002,array(4,[v00010,v00011,v00012,v00013])).
value(v00003,float(3.200000)).
value(v00004,char('q')).
value(v00005,enum(f3)).
value(v00006,pointer(v00001)).
value(v00007,structure([[(s1,v00014),(s2,v00015)]])).
value(v00008,pointer(v00007)).
value(v00009,structure([[(a,v00016),(ptr,v00017)]])).
value(v00010,integer(9)).
value(v00011,integer(8)).
value(v00012,integer(1)).
value(v00013,integer(4)).
value(v00014,integer(5)).
value(v00015,integer(-4)).
value(v00016,integer(99)).
value(v00017,pointer(v00009)).

Figure 7-2 Sample of the Symbol Table Description Language
Figure 7-3 Sample of FigEl
7.6 Graphical Description Language

Since the construction of FigEl's were done using the Prolog programming language, it would seem simple to use Prolog for graphics. However, the Prolog language does not support graphical primitives, thus the C programming language was used. The Graphical Description Language uses primitive objects as a basis for composition operations such as basic relationships, "join", and "place". Also the display must be shown onto the Unix X window system, so an interface with the X window system is included. In addition, the Graphical Description Language can be used as a separate system. Section 7.8 deals with the interface with the X window system.

7.7 Constraint Solving

There have been many methods of constraint solving. The method of constraint solving presented in this paper is compatible with any language, including C, Pascal, and Prolog. However, among these languages, Prolog was designed to be used for constraint solving, and thus is the best choice. Therefore, back tracking was used in Prolog for constraint solving.

For example, section 4.5 shows an example of a binary tree consisting of three nodes. This tree can be represented as follows:

draw_bin_tree(A, B, C)
    :- connect(A, B), connect(A, C),
        horizontal(B, C), right(B, C),
        left(B, A), right(C, A),
7.8 X window Interface

7.8.1 X window

X [Nye 90] was developed jointly by MIT’s projects Athena and Digital Equipment Corporation with contributions from many other companies.

The first and most obvious thing to note about X is that it is a windowing system for bit-mapped graphical display. That is each dot on the screen (called a pixel) corresponds to one or more bits in memory. Programs modify the display simply by writing to display memory. This system supports color as well as monochrome and gray-scale displays.

An X server controls a bitmapped screen. In order to make it easier to view and control many different tasks at the same time, the screen can be divided up into smaller areas called windows. A window is a rectangular area that works in several ways like a miniature screen. Each window can be involved in a different activity, and the windows currently in use are placed so they are at least partially visible.

The characteristics of a window are: (1) a window always has a parent window, which is assigned as the window is created; (2) each window has its
own coordinate system; (3) a window has a position, which locates its upper-left corner relative to its parent’s corner, a certain width and height of usable pixels within the border, and a border width; (4) a window has characteristics referred to as depth and visual, which together determine its color characteristics; (5) a window has a class of either Input-Output or Input-only; (6) a window has a set of attributes.

7.8.2 Interface

The X window system is a widely available windowing system. Xlib is a C language toolkit on top of the X-window system. Using these Xlib routines, a nice user interface has been developed.

The principles which should be taken into consideration while designing a user interface [Garg 89] are: (1) most of the things doable by the mouse should be doable by the keyboard also; (2) there should be feedback in response to most of the commands which the user has typed; (3) avoid flooded information.

This system was designed as two special windows. One is the Command window and the other is the Display window for 2-D objects drawing.

The Command window can be used for running gdb and typing of breaking point commands and drawing commands. The Display window is an independent window. 2-D objects are displayed graphically in this window.
The user is able to close this window. Also he can scroll left, right, up and down in this window.

The basic window functions are the following. The `<X11/Xlib.h>` file contains declarations of structure types used in Xlib functions. This routine also contains the window, display pixmap, Xsizehints, and XEvent. types `<X11/Xutil.h>` contains more structure definitions. 'Xopendisplay' routine connects an Xlib program to a server. To get window information, you can use "XGetGeometry" routine or "XGetWindowAttributes" routine. To create windows, you can use "XcreateSimpleWindow" routine. To communicate with the window manager, you can use "XsetStandardProperties" routine, "XSetWMHints" routine and "XSetWMProperties" routine. To select desired event types, you can use "XSelectInput" routine. For window mapping, you can use "XmapWindow" routine. For more information about these routines, you can use [Nye 90-1], [Ney 90-2], and [O'Rei 90].

"Draw_window" routine is used to create a window and "Open_Display" is used for drawing the 2-D objects on the X window screen.

7.9 Display

After layout has been performed, the display routine may be invoked in order to display the drawing on the screen. The display phase is responsible for mapping the device independent representation produced by the layout phase onto a particular display device. Among the tasks performed
are: (1) mapping the coordinates used internally used by the layout system to screen coordinates. (2) mapping different line and text display attributes onto the available line and text styles and colors, and (3) clipping graphical objects to fit the display window.

Since each figure element has its own local coordinate system with its position specified relative to the parent’s coordinate system, we must convert all coordinates to a "world coordinate system." A simple linear transformation can then be used to convert to the device’s screen coordinates.

The concept of a "modelling transformation" [ISO 81] is that this is a transformation applied to user coordinates to put them into a "world coordinate" system.

In order to speed up display, we note that if a figure element’s bounding box falls outside the "window," then all subelements will be left undisplayed. The following is Radack & Mateti’s display algorithm.

Algorithm Display

The input is a FigEl. Before invoking the algorithm, we set the graphical system’s "window" to the currently specified window. This establishes a mapping from world coordinates to screen coordinates to screen coordinates and sets clipping. The algorithm performs a preorder traversal of the FigEl hierarchy.

1. Invoke graphical routines to draw the background, border and label of the FigEl.
2. If the FigEl is atomic, draw the contents and return. Otherwise, perform steps 3 through 7.

3. Save the modeling transformation in M.

4. For each child C whose bounding rectangle falls at least partially inside the window, do step 5 through 6. Then go to step 7.

5. Set the modelling to M composed with a translation of (x,y), where (x, y) is the position of C specified in the FigEl.

6. Recursively invoke algorithm display on C.

7. Restore the modelling transformation to M.
Chapter 8

Conclusions

8.1 Work Completed

We have presented the design and implementation of a system for the automatic layout and display of 2-D objects. Our goal was to be able to display data on the execution programs, which are programmed with common high-level languages such as C or Pascal, without modification to the source program.

The system is divided into three phases: data or information extraction, layout, and display.

The data extraction phase is the process of converting the user’s data structures in the run time environment into standard form, called a symbol table description language. This was done through modification of the debugger Gdb which is widely used in the Unix operating system. Thus, new commands were added to Gdb which extract information from the Gdb’s symbol table, and then map that information onto the symbol table description language. The division and display of variables in cases of complex objects and many disconnected objects was solved with an extraction algorithm which solves this problem by first finding a "weakly connected component" by using a depth-first algorithm.
The layout phase deals with how to display objects on the screen. Layout is done by first constructing FigEl data structures. Since the symbol table description language’s syntax is similar to that of the Prolog language, Prolog was used. The use of Prolog gave the benefit of ease of construction as well as efficient layout using the placement-and-compute constraint solving method. Also, a general layout algorithm was given for general objects, and a special layout algorithm was given for special objects. In particular, the graph layout algorithm is important in the visualization of graphs and for design automation of VLSI.

The display phase is the drawing of objects onto the X-window screen using the layout algorithm. In order to do this, an interface was designed in C language for the X-window system, and a graphical description language was designed for the display. Also, a parser was designed to serve as an interface between the Prolog language and the C language.

This system has been implemented as an extension of the Gdb debugger for the Unix system using Prolog and C languages. Some drawings produced by the system are shown in the figures.

8.2 Future Work

This system was designed as a prototype. The following are some work that may be done onto this system.

Currently there are many proposed constraint solving and satisfaction methods. This system presents two such methods, but there is still a need for
reduction of display time, and more work may be needed in constraint solving and satisfaction methods.

This thesis presents a new graph layout algorithm. However, the graph layout algorithm changes according to the application. A graph layout algorithm which is even more general is needed.

Currently the Prolog programming language has the most efficient functions for constraint satisfaction, but the language does not have any graphical functions. In order to display objects on a screen, the C language was interfaced with Prolog, thus leading to increased computing time. Thus, if and when graphical functions are included in the Prolog language, this object display system will work much more efficiently.
REFERENCES


[Gold 83] A. Goldberg and D. Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.


[Lint 89] M. Linton,"InterViews 2.5." Stanford University, Stanford, CA 94305.


[Wall 86] V. L. Wallace, Declarative Structure in Graphics Programming Languages, Department of Computer Science, University of North Carolina.
