A Framework for Constructing Visualization, Animation, and Direct Manipulation Interfaces
視覚化、アニメーション、直接操作インターフェース作成のための枠組

by
Shin Takahashi
高橋 伸

A Dissertation

Submitted to
The Graduate School of the University of Tokyo
in Partial Fulfillment of the Requirements
for The Degree of Doctor of Science
in Information Science

November 2002

Abstract

This thesis describes a framework for developing kinds of non-WIMP-based user interface software, designated as a bi-directional translation model[1]. Here, '*non-WIMP-based*' interface means that it cannot built only by combining WIMP widgets, such as buttons and menus. In particular, the targets of this thesis are two types of GUI software. One is direct manipulation interfaces for figures and diagrams that visually represent various abstract objects and relations in an application data, such as direct manipulation interfaces for network and hierarchical data structures. They enable the user to draw and modify diagrams to input abstract data into an application. Another is animations of changing abstract objects and relations in an application data, such as animations of sorting algorithms. They show their transitions visually, and are useful for understanding the behavior of an application or an algorithm.

The bi-directional translation model models the general process of visualization, picture recognition, and animation generation. It is originated in Kamada's general framework for visualizing abstract objects and relations. We have extended it by integrating recognition of figures/diagrams and animation of changing abstract objects and relations, which is one of the contributions in this thesis. The key idea of the framework is that these functions are *translations* between different data representations. The model introduced four data representations: AR (Application Data Representation), ASR (Abstract Structure Representation), VSR (Visual Structure Representation), and PR (Pictorial Representation). In the model, visualization is a translation from AR to PR via ASR and VSR. Recognition of figures is an inverse translation from PR to AR via VSR and ASR. The essential translation is that between ASR and VSR, because there is no a priori mapping between these representations. ASR is the representation in our model that represents the structure of application data, and it does not have explicit visual appearance information. On the other hand, the purpose of VSR is to represent the high-level structures of pictures. The mapping between ASR and VSR is the essential conversion between abstract data for an application and pictorial data for presentation. Nevertheless, the structure of ASR and that of VSR are usually similar. As users understand the meaning of abstract application data via pictures, visualized pictures should *represent* abstract application data, and thus they should have a similar structure. The programmer specifies these translations by visual mapping rule sets, which are declarative rules that define mappings between ASR data and VSR data. As both structures are similar, mapping rules are usually simple.

Animations are also handled in the framework extended naturally to the time dimension. In the framework, animations are regarded as *operations* on PR that are translated from operations on AR via operations on ASR and VSR. The translation among operations is executed maintaining consistency with the mapping relations among the AR, ASR, VSR, and PR data. We have chosen an interpolation-based method for implementation of the extended framework for animations. That is, animations are generated by interpolating a sequence of pictures translated from the running application's internal data. Rather than specifying animations (motions or transformations) directly, the

[1]WIMP stands for Windows, Icons, Menus and Pointing device.

programmer specifies transitional operations, i.e., the methods used for interpolating two pictures before and after the invocation of an operation, with transition mapping rules. Therefore, animations are determined by two types of mapping rules: visual mapping rules and transition mapping rules.

In addition, we describe the systems implemented based on the framework. TRIP2 is a system that achieved the bi-directional translation between AR and PR. We applied TRIP2 to produce several examples such as the Othello application. It is implemented on NextStep using Objective-C and Prolog. To build direct manipulation interfaces using TRIP2, the programmer writes visual and inverse visual mapping rules in Prolog. TRIP2a is a tool for constructing abstract animations that depict the behavior of program executions. It is implemented by extending TRIP2 so that we can use two functions together: bi-directional translation between abstract data and pictures, and translations from abstract data into animations. The programmer can use the same visual mapping rules of TRIP2 for TRIP2a. TRIP2a/3D is the successor of the TRIP2a system that specializes in generating animations and can also handle three-dimensional representations and event-driven animations, which is useful for animating parallel program executions. In order to be able to view animations on various platforms, the animation viewers are separated from the translation module in TRIP2a/3D. The programmer writes visual mapping rules for each animation in KLIC which are compiled with translation modules to generate translators that output animation data from the log data of the application's execution. Generated animation data can be viewed with viewers on various platforms such as X-Window, MS-Windows, and Java. We describe various algorithm animations and visualizations of program executions generated with these systems.

Supporting the debugging of visual mapping rules with our systems is difficult. As a step toward solving this problem, we describe techniques for visualizing constraint systems in visual mapping rule sets. One way is to show constraint systems as three-dimensional graphs. By looking at constraint graphs, the user can see the structure of constraint systems more directly than in their textual form. Another way is to animate constrained graphical objects to show their degrees of freedom in constraint systems by using cartoon techniques. These techniques are useful for understanding and debugging constraint systems in visual mapping rules. Using these techniques, we developed a browser for constraints in VSR of TRIP systems. It is implemented using Java and integrated with the animation viewer of TRIP2a/3D.

Acknowledgments

I would like to thank Professor Akinori Yonezawa for his valuable advice, Dr. Tomihisa Kamada for his useful suggestions, Professor Satoshi Matsuoka for recommendations regarding this research and for a number of helpful comments, and Professor Etsuya Shibayama and his group for encouragement. I am also grateful to the members of the UI group, the TRIP group, especially Ken Miyashita, Ken Nakayama, and Dr. Hiroshi Hosobe for discussions, and to the members of the Yonezawa's group for their encouragement. I also greatly thank my thesis committee, Prof. Tomoyuki Nishita, Prof. Jiro Tanaka, Prof. Jun'ichi Tsujii, Prof. Masami Hagiya, and Assistant Prof. Takeo Igarashi. Finally, I extend my sincere gratitude to my parents, my brother, and my best friends for their great and constant encouragement.

Contents

1 INTRODUCTION **15**

1.1 Background 15
1.2 Goals and Approach 17
1.2.1 TRIP 17
1.3 Objectives 18
1.4 Overview of the Thesis 21

2 Related Work **23**

2.1 Introduction 23
2.2 Visualization 23
2.2.1 Information Visualization 23
2.2.2 Visual Programming Languages 24
2.2.3 Performance Visualization 25
2.2.4 Algorithm Animation 25
2.2.5 Graph Drawing Systems 26
2.3 Constraints 27
2.3.1 Constraint Solvers for GUI software 27
2.3.2 Other Systems Using Constraints 29
2.3.3 GUI Toolkits Using Constraints 30
2.4 Recognizing Figures 30
2.4.1 Spatial Parser Generators 30
2.4.2 Systems that Parse Figures 31
2.5 GUI Construction Tools 31
2.5.1 Programming by Example and Programming by Demonstration Approaches 31
2.5.2 Interface Builders 32

3 The Bi-Directional Translation Model **33**

3.1 Overview of the Bi-Directional Translation Model 33
3.2 Data Representations 34
3.3 Various Functions in the Model 36
3.4 Extended Model for Animation 41

4 TRIP2 **45**

4.1 Introduction 45
4.2 The TRIP2 System 45
4.2.1 System Overview 45
4.2.2 The TRIP Module 47

4.2.3 The Interaction Module . . . 50
4.2.4 Implementation of the Inverse Translation . . . 52
4.3 Examples . . . 56
4.3.1 A Simple Graph Editor . . . 56
4.3.2 A Simple E-R Diagram Editor. . . . 56
4.3.3 A Small Othello Game . . . 56
4.4 Related Work . . . 58
4.5 Conclusions and Future Work . . . 60
4.6 Classes in TRIP2 . . . 62

5 TRIP2a — Constructing Algorithm Animations 63
5.1 Introduction . . . 63
5.2 How to Construct an Animation — Insertion Sort Example . . . 64
5.3 Implementation of TRIP2a . . . 66
5.3.1 Implementation of Animations . . . 67
5.3.2 Specifying Transitional Operations . . . 68
5.3.3 Application Interface . . . 69
5.4 Examples . . . 70
5.4.1 Animations of Data Structures . . . 70
5.4.2 Sorting Algorithms . . . 74
5.4.3 The Tower of Hanoi . . . 78
5.4.4 Bin-Packing Problem . . . 78
5.4.5 Finding a Minimum Spanning Tree . . . 82
5.5 Incorporating Event-Driven Animations . . . 82
5.5.1 The Basic Model . . . 82
5.5.2 The Problem . . . 85
5.5.3 Approach . . . 87
5.5.4 Implementation . . . 88
5.5.5 An Example . . . 91
5.5.6 Summary . . . 91
5.6 Conclusions . . . 91

6 Visualizing and Browsing Constraints in Visualization Rules 99
6.1 Introduction . . . 99
6.2 Browsing Three-Dimensional Constraint Graphs . . . 100
6.2.1 Basic Representation . . . 100
6.2.2 Changing Layout to Explore Constraint Graphs . . . 104
6.2.3 Implementation of 3D Constraint Graph Visualizer . . . 106
6.3 Animating Freedoms in a Constraint System . . . 109
6.3.1 Overview . . . 109
6.3.2 Implementation of the Freedom Viewer . . . 111
6.4 Related Work . . . 112
6.5 Concluding Remarks . . . 113

7 Conclusions 115

A Generating Visual Mapping Rules by Examples 123
A.1 TRIP3 . . . 123
A.1.1 Overview . . . 123
A.1.2 Rule Generation for Recursive Data Structures . . . 127
A.2 IMAGE . . . 128
A.2.1 Overview . . . 128
A.2.2 Interactive Rule Generation Example . . . 129
A.2.3 Miscellaneous Issues . . . 132
A.3 Summary . . . 133

B TRIP2a/3D 135
B.1 How to Make an Animation . . . 135
B.2 How to Write a Visual Mapping Rule to Make an Animation . . . 136
B.2.1 Visual Mapping Rules . . . 136
B.2.2 The Naming of Objects . . . 137
B.3 ASR Data Representation . . . 138
B.3.1 Model . . . 139
B.3.2 Syntax . . . 139
B.3.3 An Example . . . 140
B.4 VSR Specification . . . 141
B.4.1 Graphical Objects . . . 141
B.4.2 Graphical Relations . . . 142
B.4.3 Transitional Operations . . . 144
B.4.4 An Example . . . 144
B.5 Examples . . . 144
B.5.1 N-Queen Problem . . . 144
B.5.2 The Tower of Hanoi . . . 145
B.5.3 N-Body Simulation . . . 145
B.5.4 Memory Management . . . 150

C Examples of Mapping Rules for TRIP2 155
C.1 Small Graph Editor . . . 155
C.2 Othello Game Application . . . 155
C.3 Entity-Relationship Diagram Editor . . . 160
C.4 Family Tree . . . 161

D Examples of Mapping Rules for TRIP2a 163
D.1 Data Structure Animation . . . 163
D.1.1 Graph Structure . . . 163
D.1.2 List Structure . . . 163
D.2 Sorting Algorithm Animations . . . 164
D.2.1 Bublesort . . . 164
D.2.2 Quicksort . . . 164
D.2.3 Mergesort . . . 166
D.2.4 Heapsort . . . 168
D.3 The Tower of Hanoi . . . 168
D.4 Bin Packing Animation . . . 169
D.5 Finding a Minimum Spanning Tree . . . 170

List of Figures

1.1 The visualization model of TRIP (from [67]). 19

3.1 The idea of direct manipulation interfaces. 33
3.2 A translation example in the bi-directional translation model. 37
3.3 An example of visual mapping rules. 38
3.4 Architecture of ER Diagram Editor Application. 40
3.5 The extended bi-directional translation model that represents the general architecture of applications that generates animations of program execution. 42
3.6 Circular movement of a bar in a sorting animation. 43

4.1 The architecture of the TRIP2 system. 46
4.2 A kinship diagram. 48
4.3 Visual mapping rules example. 49
4.4 Generated VSR data. 50
4.5 The execution of mapping rule. 51
4.6 The generated picture. 51
4.7 Adding a new graphical object to a relation. 53
4.8 Creating a new graphical relation. 53
4.9 Inverse visual mapping rules example. 54
4.10 The execution of inverse mapping rules that translate a set of of VSR data (`box`, `label`, and `contain`) into ASR data (`item`). 55
4.11 A simple graph editor. 56
4.12 A simple E-R diagram editor. 57
4.13 TRIP2 othello. 59
4.14 The DeltaTRIP system. 61

5.1 Annotated insertion sort program. 65
5.2 Log file of the insertion sort program. 65
5.3 A visual mapping rule for insertion sort animation. 66
5.4 A transition mapping rule for insertion sort animation. 66
5.5 Animation of an insertion sort algorithm. 66
5.6 Making an animation. 67
5.7 A cons-cell animation. 68
5.8 Slow-In-Slow-Out and Squash-and-Stretch. 69
5.9 Animation of a graph structure. 71
5.10 Allocation of a new cell. 72
5.11 Moving the substructure of a list 73
5.12 Bubble sort animation. 75

5.13 Quicksort animation. 76
5.14 Merge sort animation. 77
5.15 Heap sort animation. 79
5.16 The tower of Hanoi. 80
5.17 Bin-packing algorithm. 81
5.18 Minimum spanning tree algorithm (1). 83
5.19 Minimum spanning tree algorithm (2). 84
5.20 Sending a message. 85
5.21 The original & new models. 87
5.22 Architecture of TRIP2a/3D. 89
5.23 Screenshot of Java-TRIP2a/3D. 90
5.24 Screenshot of an animation that shows a search tree for the N-Queen problem. . . . 92
5.25 The target parallel N-queen-solving program (1). 93
5.26 The target parallel N-queen-solving program (2). 94
5.27 An example of log data. 95
5.28 The ASR data for N-queen animation. 95
5.29 The visual mapping rule set for the N-queen animation. 96

6.1 The constraint graph corresponding to the VSR data in Figure 6.2. 101
6.2 VSR data for a tree. 102
6.3 The resulting picture corresponding to Figure 6.2. 102
6.4 A connect constraint (purple box). 103
6.5 Nodes constrained by the clicked constraint are highlighted. 103
6.6 A constraint graph with regular structure. 104
6.7 A constraint graph of a tree in Figure 6.5(without line constraints). 105
6.8 A constraint graph — a `relative` constraint is set aside. 105
6.9 A constraint graph — a `relative` and two `average.` constraints are set aside . 105
6.10 A constraint graph — a `relative` and two `parallel` constraints are set aside. . 106
6.11 A constraint graph of N-body animation. 107
6.12 A constraint graph — Edges are bound up. 107
6.13 Target picture — Two quad-trees. 108
6.14 An example of translating ASR into PR via VSR. 108
6.15 Pulling a node in a tree — Movable in this direction. 109
6.16 Pulling a node in a tree — Immovable in this direction. 110
6.17 Pulling a node in a tree — All nodes are well-constrained. 110

7.1 The architecture of the next system. 120
7.2 Visualization framework with XML-VL[54]. 121

A.1 Programming by visual example — a process of generating visual mapping rules. . 124
A.2 TRIP3 — picture editor. 125
A.3 TRIP3 — confirmation panel. 126
A.4 An instance of mapping relation extracted from an example (above) and the generalized mapping relation (below). 126
A.5 A template for generating mapping rules. 127
A.6 TRIP3 — an example of tree drawing. 127
A.7 TRIP3 — ASR example. 128
A.8 TRIP3 — recursive mapping rule. 128

A.9 Interaction between a programmer and IMAGE. 129
A.10 Screenshot of the IMAGE system. 131
A.11 IMAGE — screenshots of drawing editor in rule generation for "organization". . . 132
A.12 IMAGE — organization diagram screenshot. 132
A.13 IMAGE — set diagram screenshot. 133

B.1 A mapping rule example. 136
B.2 3D tower of Hanoi. 137
B.3 Two ways of naming objects. 138
B.4 ASR data file. 139
B.5 VSR data example. 145
B.6 7 queen problem. 146
B.7 7 queen problem — distortion technique. 147
B.8 A visual mapping rule set for the seven queen problem solving animation. 148
B.9 The 3D tower of Hanoi animation. 149
B.10 Input ASR data list for N-body animation. 150
B.11 An example of visual mapping rules for 3D visualization: quad-tree (excerpt). . . . 151
B.12 An example of 3D visualization: quad-tree. 152
B.13 An example of 3D visualization: cache. 153

Chapter 1

INTRODUCTION

1.1 Background

There is a common saying that "a picture is worth a thousand words." Human beings have displayed various types of information visually since the Paleolithic age. Cave paintings at Lascaux and Altamira indicate that humans have always drawn pictures, which serve as messages for communication. During the medieval period, pictures were used to represent divine concepts, and were not merely copies of real scenes. People often draw a rough sketch to represent their thinking, and these sketches represent visual communications to themselves.

The importance of visual display is increasing in the computer age. Powerful computers and high-speed networks are producing vast amounts of diverse information that is difficult and time-consuming to retrieve and understand. Visualization is one technique that can be used to alleviate this problem. With well-drawn graphs, such as scatter plots and line charts, users can more easily understand statistical information. The results of large scientific computations cannot be fully understood without accompanying sophisticated visualization. UML[1] diagrams are widely employed to visualize complex structure of large software. Visualization is, therefore, an indispensable technique for modern computer systems.

Pictures are helpful not only for presenting information to the user, but are also useful for the user to provide input to computers. Graphical user interfaces (GUIs) enable users to input information visually. Computers show visual representations of various data to the user, who edits the visual representations to change the corresponding data. Users actually feel as if visual representations are the data, and think that they are handling the data directly. This is an instance of technique called *direct manipulation*(DM)[109]. Shneiderman pointed out four principles of DM interfaces in [109]:

- Continuous representation of the object of interest.
- Physical actions or labeled button presses instead of complex syntax.
- Rapid, incremental, reversible operations whose impact on the object of interest is immediately visible.
- Layered or spiral approach to learning that permits usage with minimal knowledge.

The concept of direct manipulation is widely incorporated into current GUI software. In particular, it is essential for the applications that handle visual information such as figures, diagrams, and pictures.

[1] Unified Modeling Language. For details, see `http://www.uml.org/`

One of the goals of user interface research is to allow users to communicate with computers in diverse ways, including DM-style interfaces. Which method is better is determined by its purpose. The user interface should be chosen so that it is appropriate for the information that it handles. Most traditional communication between users and computers has been performed through character-based methods. The user inputs commands with a keyboard, and the computer presents its output in the form of text. This style of user interface is still dominant in most current UNIX systems. Expert users love such systems, and they are indeed useful for textual tasks, such as writing documents and developing programs. However, such command-line interfaces are not suitable for the communication of visual information. GUIs enhance visual interactions between users and computers. Computers show information visually, and users see and react to these visual representations with a keyboard and a pointing device. However, current GUIs are insufficient in some important aspects. Although representation of data is visual with a GUI, users cannot freely input visual information. Most current GUI systems are WIMP-based; that is, they consist of **W**indow, **I**con, **M**enu, and **P**ointing device. In a WIMP-based GUI, basic user inputs are selection and clicking of menus, icons, and buttons using a pointing device and input from a keyboard. They are limited in the sense that users do not easily input visual data such as pictures, figures, and diagrams.

There are some *visual* applications. For example, drawing and painting software enable users to create figures and pictures with a pointing device. Designers use CAD systems to design architecture, industrial products, electronic circuits, etc. In visual programming environments (VPEs), programmers can construct programs visually and interactively. However, these systems have not yet reached our goal of allowing users to communicate with computers in diverse ways. Drawing editors cannot understand the pictures that users draw; that is, drawn pictures are treated as images from which systems cannot extract meaning. CAD systems and VPEs recognize drawn figures, which are used for input from users. In this sense, these systems are communicating with users via visual representations of data. However, the syntax of the pictures that they can understand is rigid and built in; it is difficult to provide software that can recognize and understand figures with user-defined syntax. More flexible visual communication between computers and humans, and better communication among humans with the assistance of computers, are desirable goals for computer-human interaction. In addition, computers would be able to assist drawing work intelligently if they could understand what the user is drawing.

Various tools have been developed to ease the burden of visualizing information. Scientific visualization tools such as AVS[30] and Khoros[4] are widely used by physicists for visualization of large numeric data, such as the results of fluid mechanics' simulations. Spreadsheet applications that can produce business graphics easily from a table are another widespread example. Mathematica[129] and MATLAB[84, 35] are powerful environment that helps scientific calculation and visualization of the result. However, they basically provide ready-made visualization libraries for numeric data. Although it may be possible to customize them by changing various parameters, it is difficult to implement user's own visualization methods. In addition, they are not designed to handle relational and structural data.

The visualization of dynamic information is referred to as "animation." In particular, *algorithm animation*, animations that depict the behavior of algorithms, is one highly active area of user interface research. *Sorting Out Sorting*[6] is a well-known example that shows animations of various sorting algorithms. A number of systems to support the development of algorithm animations have been proposed, such as BALSA[15], Zeus[16], Tango[116], and Pavane[103]. However, these are tools for making devised animations. It is still a difficult and tedious task to make animations for personal purposes, such as verifying or debugging the behavior of programs.

1.2 Goals and Approach

To approach this ultimate goal, support software for constructing such user interfaces must be improved. Libraries for graphical user interface (GUI) software allow their more rapid development with less effort. Although current GUI libraries and tools are useful for the development of WIMP-based interfaces, they are of no use for making more interactive parts of user interfaces. Most programmers still write these from scratch using primitive library functions.

The objective of this study is to reduce the cost of developing GUI software. In particular, we aim at two types of interfaces: One is direct manipulation of pictures, figures, and diagrams that visually represent the structure of abstract data in an application. Another is animation of changing abstract data in an application which shows transitions of abstract objects and relations visually. They are not merely an aggregation of buttons and menus, and they cannot be developed only by combining WIMP widgets provided by user interface toolkits such as Java Swing set[125] and GTk+[98]. Many researchers in this area are addressing this problem, and a number of techniques have been developed for this purpose, including Programming by Example (PBE) and Programming by Demonstration (PBD). Garnet[91] and its successor, Amulet[14], are well-designed toolkits for GUI construction. These systems incorporate a number of innovative techniques and ideas. However, they are insufficient, in that they do not support the development of user interface software that can facilitate flexible and intelligent visual communication.

Our approach is, first, to create an integrated framework for visualization, recognition, and animation. This framework decomposes the processes of visualization, recognition, and animation into several sub-processes. We can implement visualization, recognition, and animation programs by developing modules that achieve each sub-process. The merit of creating a framework is that it avoids ad hoc construction of user interface software. We develop the integrated framework by extending Kamada's general framework for visualizing abstract objects and relations[65, 67], which is the model of the TRIP system. TRIP realized general visualization of abstract objects and relations based on its framework for general visualization. It regards the visualization process as the *translation* from application's data representations into pictorial representations. However, it handles only visualizations of abstract application data. This thesis extends their framework to accommodate recognition of drawn figures and animations that depict change of abstract application's data.

Second, we examine whether each functional module in the framework can be shared among different instances of GUI software. Sharable modules can be built in advance and provided as a library/tool for building GUI software. The essential part of the GUI software cannot be shared, but we have made it possible to build with simple specifications. Third, based on this framework, we have built tools that support the construction of GUI software. We have built several systems (TRIP2, TRIP2a, TRIP2a/3D), and applied them to the construction of various applications. Using these tools, users can construct GUI software by providing only declarative specifications. We have also developed a tool for visualizing the structure of constraints in the specifications, which is helpful to understanding and debugging the specifications.

1.2.1 TRIP

As described above, our integrated framework is originated in Kamada's general framework for visualizing abstract objects and relations[65, 67]. We have extended it by integrating recognition of figures/diagrams and animation of changing abstract objects and relations. This section briefly describes Kamada's framework, and the prototype system TRIP developed based on the framework.

The objective of their work is to fill the gap between application and graphics systems that is usually only a set of drawing instructions. There were few graphics tools that are helpful to

visualize complicated high-level abstract relations in an application into pictures that represent their structures. TRIP and its framework intended to explore a general approach to the visualization of abstract objects and relations, and help visualizing abstract application data.

The framework models the visualization process as the *translation* from textual languages into pictures. A set of pictures can be viewed as a visual language, in the sense that picture elements are arranged under certain layout rules. Thus, visualization is the translation from textual languages into visual languages. When translating textual data into pictorial data, the programmer should determine the layout of pictures so that viewers can easily understand the underlying structures. The programmer needs to map a relational structure in textual data into a layout rules of picture elements. We call this mapping as *visual mapping*. In the framework, application data are represented by abstract objects and relations, which are mapped to graphical objects (such as boxes and circles) and graphical relations (such as horizontal/vertical listings and line connections). The mapping is specified not with procedures, but with declarative mapping rules. Thus, the programmer can easily change the layout of pictures by changing declarative mapping rules.

TRIP (TRanslation Into Pictures) is a prototype system based on the visualization framework. It is independent of both input textual languages and output visual languages and can handle various visualization problems. The process of visualization proceeds as follows (Figure 1.1):

- First, an analyzer parses original textual representation data, and generates the relational structure representation data. Analyzers depend on the textual representation, because it needs syntax data of original textual representation for parsing them. Parser generators such as YACC[77] can be used to help developing analyzers.

- Second, relational structure data are mapped to visual structure data. Abstract objects are mapped to graphical objects, and abstract relations among them are mapped to graphical relations among corresponding graphical objects. In TRIP, relational structure and visual structure are represented in Prolog. Abstract relations and objects are represented by Prolog predicates. Graphical relations and objects are special predicates that are defined in COOL — a picture layout module in TRIP.

- Finally, actual layout of graphical objects are computed by solving graphical constraints and a picture is generated. They basically handles diagrams structured by horizontal/vertical listings and explicit line connections. They developed COOL(COnstraint Object Layout system) to layout picture elements. It solves constraints and outputs pictures. It can handle over-constrained equation systems by using least squares method. It also has general undirected graph drawing module, which is useful for visualize network diagrams. The graph layout module uses the spring-model algorithm[66].

Using TRIP, they generated various diagrams from abstract textual data. For example, they generated kinship diagrams from English sentences that states kinship relations, list diagrams that show cons-cells from S-expressions, and E-R diagrams from ER schema. In addition, they have shown that the programmer can generate various types of tree diagrams from the same tree data only by changing the mapping rules.

1.3 Objectives

The objectives of this thesis are as follows:

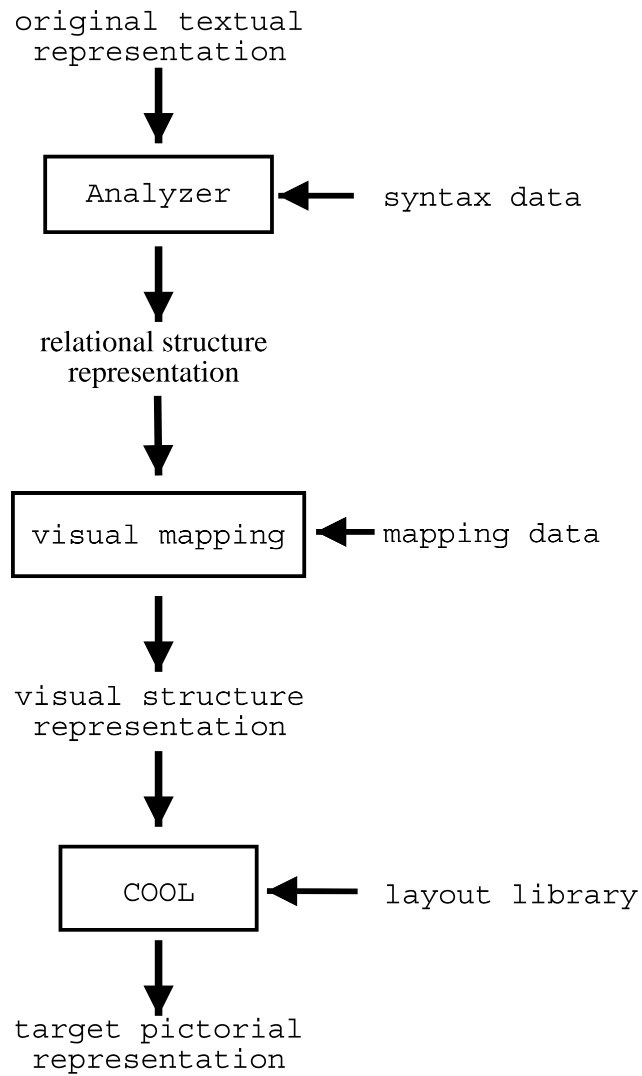


Figure 1.1: The visualization model of TRIP (from [67]).

An Integrated Framework

The first objective of this study was the development of a general framework for constructing visualization, direct manipulation, and animation interfaces. This framework describes the structure of these user interfaces, and supports the development of toolkits for building direct manipulation and animation interfaces. According to this framework, these interfaces consist of modules that translate four types of data representations, i.e., application data representations, two intermediate data representations, and pictorial representations, into the other types of representation. Thus, the development of these user interfaces allows implementation of translators among four data representations. Essential translators are those between two intermediate representations. These must be created by the user for each target application, but they can be built by writing only declarative mapping rules that specify the translations. Other translators are common among different targets, so they can be built as a library and provided for the user.

The basis of this framework is the model of the TRIP system[67], which handles the visualization of abstract data, and is regarded as a translation from abstract data into pictures. Our framework extends this to handle the recognition of pictures and animations, which are also regarded as translations between data representations in the framework.

Tools implemented based on the framework

The second objective of this study was the implementation of three tools based on the above framework. One is the TRIP2 system, a tool that can visualize abstract data as a figure, and can also recognize modifications performed on the visualized figure. This function can be regarded as a kind of direct manipulation of abstract data, which is achieved by bi-directional translation between abstract data and figures specified by a set of visual mapping rules and a set of inverse visual mapping rules. The other two tools are the TRIP2a and TRIP2a/3D systems for making animations that represent the execution of the programs. In our framework, animations are also achieved by translations generated by translating a series of changing abstract data into a corresponding series of changing pictures. Therefore, animations can also be specified with declarative rules, i.e., a set of visual mapping rules and a set of transition mapping rules. TRIP2a was developed by extending TRIP2, and it is integrated with the TRIP2 system. Therefore, both direct manipulation of abstract data and animation of changing abstract data can be utilized to develop interactive applications that show algorithm animations. TRIP2a/3D is another algorithm animation system, which is a re- developed system for animating the execution of programs. This system has been extended to handle three-dimensional animations and a type of event-driven asynchronous animation.

With these tools, we can create applications that have a kind of direct manipulation user interface and animations by providing only declarative mapping rules. These tools can reduce the cost of implementing interactive user interfaces.

Methods for Visualizing Constraints

The third objective of this study was the development of a new method for visualizing constraints in the visual mapping rules. Although this method was developed mainly for debugging visual mapping rules that specify the translations among four data representations, it can be applied for visualization of general constraints. We have developed a visualization system using this method. This system visualizes constraints in two ways. The first is to visualize them as a three-dimensional graph structure, with nodes in the graph as constraints and constrained variables. Edges connect a constraint and the variables by which it is constrained. By showing a constraint system as a graph structure, the viewer can grasp the overall structure of the constraint system. It is also possible to

unfocus some constraint nodes in the graph so that its structure becomes simple. It helps the user to browse the constraint graph. The other uses our new method, which animates the degree of freedom in the under-constrained system. This method displays the target figure itself, and tries to 'move' each graphical object in the figure, which may have various constraints. If the object is sufficiently constrained, the system will not be able to move it, while if it is not constrained and has some degrees of freedom, the system will be able to move it. Therefore, when there are some degrees of freedom, the system shows an animation in which objects are moving in some way. Otherwise, the system shows an animation indicating that the object cannot move. This animation can help users to readily recognize any lack of constraints.

1.4 Overview of the Thesis

In this thesis, the framework for visualization, recognition, and animation is first presented. Second, several systems based on the framework are described, and the framework is verified based on knowledge acquired through construction of the above systems.

Chapter 2 surveys various work in related research areas; i.e., visualization, user interface toolkits, algorithm animation, and constraints. Chapter 3 describes the basic and extended frameworks. The basic framework handles the visualization of abstract data and recognition of visualized figures, and the extended framework handles animations. Chapter 4 describes TRIP2, an environment for building direct manipulation interfaces of abstract data. By providing only a set of visual and inverse visual mapping rules, the user can visualize abstract data, modify the visualized figure, and reflect the modified figure to abstract data. Chapter 5 presents TRIP2a, a system for animating the changing internal data during the execution of programs. The user can create animations by specifying a set of visual mapping rules and transition mapping rules. Various examples of animations created with TRIP2a are presented in this chapter. Chapter 5 also describes an extension of TRIP2a that is capable of handling asynchronous motion in animations. This was achieved by slightly modifying the framework to accommodate start and stop events of motion. In addition, as an appendix, we describe TRIP2a/3D; i.e., a tool for making three-dimensional algorithm animations. Several sample animations generated with TRIP2a/3D are also presented. Chapter 6 describes a new visualization method and the tool for visualizing constraints in the VSR generated by visual mapping. This is useful for debugging visual mapping rules used to generate the VSR from ASR data. Finally, Chapter 7 summarizes this thesis and its conclusions, and proposes some directions for future work in this field. The appendix also introduces the TRIP3 system and the IMAGE system; i.e., work on the interactive generation of visual mapping rules for TRIP systems, described here because it is closely related to the work discussed in this thesis.

Chapter 2

Related Work

2.1 Introduction

This chapter describes research into support software for building graphical user interfaces, especially GUIs that cannot be built only by composing WIMP widgets, and is organized into four sections: visualization, constraints, recognizing pictures, and GUI toolkits.

In the *Visualization* section, various areas of visualization software are summarized. The main concerns are first how to effectively present various types of information to the user, and second how to reduce the cost of implementing such visualization. This section includes sub-sections on, for example, information visualization, performance visualization, and algorithm animation. Graph-drawing systems are also described in this section.

Constraints are among the most important techniques in GUI software. The Constraints section describes user interface software systems using constraints, and constraint solvers for GUI software.

The *Recognizing Figures* section summarizes spatial parser generators and systems using spatial parsing, i.e., modules that parse drawn figures according to the specified syntax of two or three-dimensional figures. Although the range of figures that they can handle is limited, spatial parsers are useful for implementing applications that accept drawn figures as their input.

The section on *GUI Construction Tools* summarizes research on software and techniques that help in the development of various types of graphical user interface software. The main topics are Programming by Example-based systems and interface builders.

2.2 Visualization

2.2.1 Information Visualization

Information visualization is a research area that studies how various types of information can be presented effectively to the user. It facilitates the navigation of large information spaces and extraction of hidden structure and meaning. One of the most important issues in this area is scalability, that is, how to handle large amounts of information.

One approach to address this issue is to use three-dimensional representation. By using three-dimensional display, one more dimension can be exploited to devise the representation. For example, in the 3D/Rooms of Information Visualizer[19], the user can visit a number of rooms in which various types of three-dimensional figures are floating. For example, the Cone Tree[102] is a three-dimensional tree that represents hierarchical information. The user can click the node in the tree so that the tree is rotated and the clicked node is moved to the front. These rotations are animated

smoothly to prevent the user from losing track of the node. Subtrees can be displayed as translucent cones to hide unimportant nodes. The Perspective Wall is another figure in 3D/Rooms, which shows calendar-like information on a curved wall. The center of the wall is placed near the user, so it is magnified in the display, and the two sides are presented diagonally to the user, so areas closer to the ends of the wall are displayed as smaller on the screen.

Another approach to this issue is zooming. In zooming interfaces, everything is placed on one large plane, and the displayed view is a magnified, focused, area of the plane. The user navigates through the plane by zooming in/out with a moving focus point. The distortion and fisheye view techniques are similar to zooming. The basic idea of these techniques is that everything should be displayed on the screen, the focus point should be magnified, and other areas should appear diminished in size. As the focus point is always surrounded by the diminished area that serves as its context, the user can easily find and move to the target point in the search space. The most well known research into the use of the fisheye-view technique is the work of Furnas[43], who visualized hierarchical data structures using this method. Koike[71] presented a fractal view technique that can control objects to be displayed in average. Continuous zoom[7] is another zooming algorithm that handles a rectangular 2-D display space.

Another issue of information visualization is how to support design tasks for visualization. Visual design of effective information presentations is not easy for most users, and a great deal of work is required to support these tasks. For example, Mackinlay developed APT[79], which can automatically design effective graphical presentations such as bar charts and scatter plots of relational information. In this work, he codified the graphical presentation technique and developed *composition algebra* that can form complex presentations from that technique. SAGE[104] systems are tools that interactively support graphical design. SAGE can be used for designing according to partial user specifications that serve as design directives. It is also possible to draw and combine graphical elements interactively, and to browse and retrieve previously created pictures for use as design examples.

It is also important to enable users to interact with visualized information. Dynamic query interfaces[2] enable the user to control the range of information displayed with various tools. One such tool is a range slider that can specify the range of values easily, which is useful for filtering the information to be displayed. The results of making changes to the tools are instantly fed back to the display so that the user can easily access the desired information.

2.2.2 Visual Programming Languages

Most visual programming languages have a non-textual visual representation of programs; i.e., the programmer can *draw* programs in visual form. For example, Prograph[29] is a commercial dataflow-driven visual programming language. Its program is a diagram that consists of icons that correspond to operations and links that connect icons. CafePie[96] is a visual programming system for CafeOBJ[34]. It visualizes the program and its execution with animations. The user can customize its view interactively.

The merit of visual programming languages is that they can depict the overall structure of programs naturally. The relations among program modules become easier to understand, in the same way as CASE tools. However, they are not always suitable for programming small details because they require such a large space for drawing programs and it is often more tedious to draw programs with a mouse than it is to write using a keyboard.

2.2.3 Performance Visualization

Performance visualization aims to provide performance data regarding program execution in various ways. The user examines the visualized information for debugging and performance tuning.

Performance visualization is particularly important when using parallel programs, the behavior of which is difficult to understand. ParaGraph[78] is a system for performance visualization that handles parallel programs in a message-passing multi-computer architecture. ParaGraph visualizes trace data monitored during actual program execution. Program executions can be replayed pictorially based on trace data. In addition, various aspects of performance data can be extracted from trace data and visualized appropriately. Pablo[100] is a similar performance visualization system for massively parallel architecture, which has been extended to SvPablo[33]. Paradyn[86] is also a performance visualization system for parallel and distributed systems; it can identify possible performance bottlenecks in program execution.

2.2.4 Algorithm Animation

Algorithm animations are animations that depict the behavior of algorithms. The main purpose of algorithm animation is education. A previous study has shown that algorithm animations are useful for learning algorithms[118].

Sorting out sorting is one of the most impressive algorithm animations, which first explains the usefulness of algorithm animations. It animates several sorting algorithms simultaneously, which greatly assists the user to compare the behavior of each algorithm. The process of sorting is animated as moving points that correspond to the elements being sorted. The y-coordinate corresponds to the value of the element, and the x-coordinate corresponds to the position of the element. The points are finally placed in a line. The changes in the distribution of points indicate the characteristics of the sorting algorithm.

Most subsequent algorithm animation systems are environments to facilitate the creation of algorithm animations. The best-known systems in the early days were BALSA and BALSA-II[15]. These systems have been applied to many algorithms, such as sorting algorithms, bin-packing algorithms, and many types of graph algorithms, and they have been widely used by students and researchers. The concept of *interesting events*, which inform the animation system that an ”interesting” operation has been executed, was introduced by the BALSA system. Zeus[16, 17] is the successor of BALSA II, which is implemented on Modula- 3. However, neither BALSA-II nor Zeus directly supports the creation of the graphics parts of animations in a declarative way.

TANGO[115, 116, 114] is based on a *Path-Transition* paradigm. In this model, there are four data types: location, image, path, and transition. An animation consists of transitions comprised of pairs of images and paths. An image is an object drawn on a screen, and a path describes a series of locations. TANGO also uses interesting events for calling these animations. The path can be generated by the user’s demonstration using the Dance system[117]. However, we believe that it is more laborious to specify complex animations involving compound data structures with TANGO than it is with our system. Polka[61] is the successor of TANGO; it allows the creation of 3D animations of concurrent programs.

Pavane[28, 103] is a declarative animation system. This system visualizes programs written in the *Swarm* language, which is based on the state-transition model. Animation is specified by declarative mapping from *State* space, which is a set of tuples that represent the data of the program to be visualized, to *Animation* space, which contains graphical objects with time-dependent attributes. The difference between this and our model is that our method of specifying an animation is more abstract, and is thus more concise. For example, using our system, the programmer can specify

the motions of objects, including animation effects, by simple transitional operations. In addition, the layout of a picture can easily be specified using graphical constraints in our system, which also makes it easy to specify an animation.

Animus[36] uses *temporal constraints* for creating animations. These are specified in the same manner as with ThingLab[11] and ThingLabII[80]. In Animus, the appearance and structure of a picture, as well as how pictures evolve over time, are described by constraints. Dynamic mechanical and electrical circuit simulations, problems in operating systems, and geometric curve-drawing algorithms have been animated using Animus[36].

Pictorial Janus[63, 64] is a visual representation of Janus, a general-purpose concurrent constraint programming language. It is *complete*, which means that pictures are programs and represent screenshots of computation, and it completely corresponds to the text version of Janus. In addition, the execution of programs (pictures) can be animated.

Bentley and Kernighan developed ANIM[9, 10], which aims to allow easy creation of animations. In this system, an animation is made from a *script file*, which is output from a target application program that contains commands for drawing an image, such as `box` and `line`.

Some systems incorporate the techniques of cartoon animation[76]. The user interface of Self[20] utilizes such techniques for the movement of windows, and Artkit[108] provides animation abstraction, which also supports cartooning techniques.

2.2.5 Graph Drawing Systems

Graph-drawing produces a nice layout of a given graph $G = (V, E)$ (where V is a set of vertices and E is a set of edges). The problem is formalized as positioning vertices $p \in V$ by determining the mapping $\phi : V \to R^2$ (or R^3). Edges are straight lines among the positioned vertices. However, when applying them to real applications, vertices may have size and occupy some area, and edges may bend or curve.

Graph-drawing algorithms can be classified by the types of their target graphs,which have different criteria for appropriate layout. For trees, a number of algorithms to minimize the area required to draw a tree have been proposed. Algorithms for planer graphs draw figures without edge-crossing. Directed graphs are drawn so that the edges are oriented in one direction. Graph-drawing algorithms are summarized in the annotated bibliography of Tammasia [8].

General graph drawing is much more difficult than specific graph drawing because no assumptions about the structure are possible. In addition, simple criteria are often not enough to allow the drawing of general graphs. For example, there are cases in which minimizing edge-crossing is not natural, and it is necessary to maintain the balance of various criteria.

One of the most important approaches for general graph drawing is the force-directed approach in which nodes in a graph are treated as material points. Pulling and repulsing force are exerted on nodes. For example, in the spring embedder[37], each edge in a graph is treated as a spring. This system places nodes at random positions, and simulates their movement. As it is assumed that there is friction, they eventually reach the state of minimum energy. In Kamada and Kawai's algorithm[66], every pair of nodes in a graph is connected by a spring the length of which is equal to the graph- theoretical distance between the two nodes. The layout is obtained by moving each node one by one to minimize the overall energy of this dynamic system.

With these force-directed approaches, general undirected graphs can be drawn quite well. However, these force-directed algorithms are very slow and do not scale well. Drawing large-scale general graphs, such as graphs with over 1000 nodes and edges, is difficult.

The multi-scale approach recently proposed by Hadany and Harel[46] and in subsequent studies by Harel and Koren [47] promises to target large graphs such as those with over 3000 nodes. Their

key idea is a coarse-scale representation of the graph that represents its simplified structure. This representation consists of clusters of nodes and edges between clusters. The algorithm lays out fine-scale positions of nodes locally in a cluster, and then arranges coarse-scale positions of nodes globally. They claim that a graph with over 3000 nodes can be arranged in a few seconds using this system.

In real applications, it is often necessary to reflect the user's preferences. There are several graph-drawing tools in which the user can select an appropriate drawing algorithm. These include GraphEd[50], Gem[1], GraphViz[38], or Graph Editor/Layout Toolkit by Tom Sawyer Software[112].

Henry and Hudson[49] proposed a graph-drawing tool that can apply different algorithms to subgraphs in a graph. The subgraph is interactively selected by a specific graph-traversing algorithm. For example, the user can select nodes on the shortest path in a tree, and arrange them in a row. The other nodes in a tree are then drawn using an ordinary tree layout algorithm. The merit of this approach is that it is possible to handle very large graphs by selecting and drawing only subgraphs.

GLIDE[68] is another system in which the user can interactively lay out graphs. The user can specify various constraints among nodes. For example, the user can specify that selected nodes should be aligned horizontally or positioned circularly. All constraints in GLIDE are solved by a force-directed approach, and the process of solving constraints is animated so that the effects of constraints can be well understood by the user. Constraints are regarded as various 'springs' among nodes. Therefore, the user can specify conflicting constraints that can be solved approximately by balancing these constraints.

Masui[83] proposed a graph-drawing system based on a genetic algorithm. The user provides the system with several pairs of examples, and the system infers the evaluation functions using the genetic programming technique.

ILOG JViews[106] is set of Java components for building interactive graph-based user interfaces. It provides various components for cartography, charting, resource scheduling, and graph-layout. The method of visualization can be specified using XML style sheets.

2.3 Constraints

2.3.1 Constraint Solvers for GUI software

Constraint-solving problems are studied widely in various areas such as artificial intelligence and operations research. Various resource allocation problems and task scheduling problems are typical examples of constraint-solving problems. The merit of using constraints is that programmers have only to write the problem to be solved declaratively and do not need to specify how the problem should be solved. The constraint solver solves the specified problems automatically.

Constraint solvers have also been applied for construction of graphical user interface (GUI) software. For example, they are useful to determine the layout of parts of the GUI on the screen, and to reflect the application's internal changes/events on the screen. Several constraint solvers have been developed for GUI software. As they were developed for interactive GUI software, they have different characteristics from those developed for other areas. The characteristic functions required if constraint solvers are to achieve interactive systems are:

Preference A typical use of constraints for constructing GUI software is to specify and determine the layout of graphical parts. For example, the layout of a slider S placed at the right side of

a window W can be specified with a constraint:

$$S.left_x = W.right_x$$

A constraint specifying that the heights of the two windows are the same is:

$$S.height = W.height$$

$S.left_x$ is the x coordinate of the window S, and $W.right_x$ is the x coordinate of the window W. $S.height$ and $W.height$ are the heights of each window. Most constraints to determine the layout of widgets are required to be satisfied.

There are also esthetic criteria that must be considered when creating a picture. The programmer must manage screen space, and use constraints to place objects neatly. For example, any lines connecting objects should not cross each other. Graphical objects must not be too close, and the distances between objects should be even.

However, it is not always possible to satisfy such criteria, and it is not always necessary to satisfy all of these criteria at the same time; i.e., there are *preferences* for constraints. Therefore, it is necessary to provide a way to specify and handle such preferences.

In the TRIP system, two levels of constraints can be used: *pliable* and *rigid*. Rigid constraints are solved exactly, before pliable constraints are solved approximately, i.e., rigid constraints have precedence over pliable constraints.

In the IMAGE system, the DETAIL constraint solver is used. The DETAIL solver incorporates constraint hierarchies[12]. That is, each constraint has a strength value, and the solver satisfies as many of the higher valued constraints as possible. DETAIL can solve constraint hierarchies incrementally, using local propagation, and still handle cyclic constraints[55].

Constraint hierarchies are also useful for specifying the dynamic behavior of interactive systems. For example, the behavior of a rectangle when the user points to it, i.e., whether it should be resized or moved, can be specified by constraints on the points: the width and the height, each with a strength value. To move a rectangle, the value of the constraints that maintain its width and height should be increased. On the other hand, to resize a rectangle, the constraints that prevent a diagonal point from moving should be given higher values. The system only changes the values of constraints according to the operation specified by the user.

Furthermore, constraint hierarchies are useful for handling over/under-constraint systems. In general, for interactive applications, the values of constrained variables should always be determined. However, solving over/under-constraint systems is a difficult problem, and it is generally difficult for programmers to avoid such systems, i.e., to specify constraints on variables with sufficient precision to determine the values of the constrained variables. By using constraint hierarchies, the programmer is able to use weaker constraints to prevent under-constrained systems, thereby assuring that constraints are always solvable.

Performance Where constraints are used to determine the behavior of graphical components on the screen, as described above, they must be solved quickly, as this influences the smoothness of the interface, and affects its usability.

There are several ways to cope with this problem, one of which is to solve constraints *incrementally*. Constraints used in interactive systems may become numerous, but they tend to be clusters of independent constraints. Therefore, once all of the constraints are satisfied, constraint solvers only need to maintain those that are affected by changes made to the

constraints. DeltaBlue[41] is a well-known incremental constraint solver that uses local propagation, and solves non-cyclic linear equations with strength values. We have developed the DETAIL constraint solver[55], which is also an incremental constraint solver. DETAIL can have multiple types of sub-solvers, such as a simultaneous equation solver and a least squares method solver, and can solve multiple types of constraint.

HiRise[51] is another incremental constraint solver, which solves constraints utilizing LU decomposition so that it is able to solve thousands of constraints in a few milliseconds. HiRise can also handle inequality constraints.

Complexity Linear equations are not sufficient to describe the behavior of interactive systems. There are more complex types of constraint, which are useful for describing various behaviors in interactive systems. For example, a general constraint that two lines should be parallel is not described by linear constraints. A constraint that maintains the distance between two points is also not linear[1].

Inequality constraints are also useful for GUIs. For example, the constraints that some elements must be put inside a window are represented as conjunctions of several inequality constraints:

$$win.left_x \leq obj.left_x \tag{2.1}$$

$$obj.right_x \leq win.right_x \tag{2.2}$$

Cassowary and QOCA[13] can handle inequality constraints. They solve constraints as an optimization problem using the simplex method. In addition, constraints that objects do not overlap are represented as disjunctions of the above constraints. Such disjunctive constraints are also described by Kim Marriott et al. [69]

Chorus[53, 52] solves non-linear constraints using the optimization method. To cope with the local minimum problem, it also uses the genetic algorithm. Chorus is so expressive so that it can specify force-directed graph layout algorithms. In addition, the solver is fast enough to drag and modify the layout of a graph interactively.

However, in general, expressive constraints often require inefficient constraint-solving algorithms, such as the Newton-Rhapson method and the relaxation method. This is why most of the constraint solvers in use in current interactive systems solve only one-way constraints using local propagation. The DETAIL solver copes with this problem by dividing constraint systems into those sets of constraints that can be solved by local propagation and those that cannot be solved. DETAIL then applies inefficient constraint-solving algorithms only to those parts that actually require them. Such combinations of different constraint solvers are promising approaches for interactive systems.

2.3.2 Other Systems Using Constraints

Constraints are used not only to support the construction of GUI applications but also as interaction techniques in GUIs. Most drawing editors have alignment and grid functions that can be thought of as types of constraints. In some editors, users can use connection constraints in which a line connected to an object follows that object when it is moved. IntelliDraw[27] is a commercial drawing application that has many constraints functions. Various types of geometric constraints are provided

[1] Of course, it is possible to specify that a line should be parallel to an axis, or to specify that the horizontal/vertical distance between two points should be constant.

for the user. However, IntelliDraw has not met with much success; one reason is that it is difficult to specify constraints directly in this application; there are too many icons representing the types of constraints from which the user can select. It is also difficult to combine constraints and to predict the effects of doing so. Furthermore, ordinary drawing may not require complex constraints.

The key issue of using constraints in a GUI is providing an easy and intuitive way to specify constraints. *Inferring* is a common technique used to address this issue. The user does not specify constraints directly but shows the state that satisfies the constraints the user intended. The system then infers constraints from the presented state. For example, Chimera[72] infers constraints in a picture from multiple examples of pictures. Characteristics that are invariable among examples are regarded as constraints.

For example, the constraints of a hinge, two lines that are connected at one end of each line and can be rotated at that point, can be extracted from several examples of a hinge with different angles. Pegasus[56] is another pen-based drawing tool that infers constraints in a picture. In Pegasus, the user can draw only straight lines, but the application can recognize several relationships among lines such as parallel, perpendicular, same length, and symmetry. Pegasus infers lines to be drawn based on recognized constraints, and presents them to the user. The user can select one from this list or can neglect them by drawing another line.

2.3.3 GUI Toolkits Using Constraints

Constraints are incorporated into several GUI toolkits. Garnet[91] and its successor, Amulet[14], have a simple one-way local propagation-based constraint solver. The programmer can define constraints among variables in the program so that when the values of source variables are changed, the target variables are updated appropriately.

Various toolkits use constraints for the layout of components in GUIs. SubArctic[107] incorporated a one-way constraint solver for the layout of components in a GUI. *Open Inventor*[126], which is a three-dimensional graphics toolkit, can have nodes called *engines* in the scene graph, which also correspond to one-way constraints that connect attributes of other nodes. Recent GUI toolkits have a mechanism of layout management such as the layout managers in Tcl/Tk and Java AWT. These can be thought of as special constraint solvers that solve predefined layout constraints on target widgets.

2.4 Recognizing Figures

2.4.1 Spatial Parser Generators

Spatial parsing is a function that parses pictures drawn by the user according to the grammar specified by the programmer. The *spatial parser generator* generates *spatial parsers* from the grammar specified by the programmer. Ordinary drawing editors with spatial parsers can be used as interaction modules for visual language systems.

Spatial parsing has been studied by a number of researchers for a long time. For example, Chang proposed "picture-processing grammar" [22]. Lakin developed the PAM system and the VMACS editor [75]. VennLISP[73], a visual notation of LISP, is one application of these systems. Various grammars have been proposed. For example, Wittenburg proposed relational grammar[127] and bottom-up and Earley-style parsers. Rekers proposed the use of graph grammars for graphical languages[58], which were later refined to layered graph grammars[101]. Constraint multi-set grammar (CMG)[82] is another well-known example of grammar for parsing two-dimensional figures. The production rules of CMG have the form $P ::= P_1, P_2, ...P_n$ where C, which means that

$P_1, P_2, ..., P_n$ can be reduced to the non-terminal symbol P whenever the attributes of all symbols satisfy the constraint C. Terminal symbols represent primitive shapes such as lines and boxes. Eviss[5] and its successor Rainbow[62] is an example of a visual system that uses CMG.

2.4.2 Systems that Parse Figures

There are many systems that can recognize figures drawn by the user, and utilize them as input to the application. Most visual language systems have built-in parsers that interpret visual programs. An interface builder is a type of VL system. Programmers can lay out widgets visually on the screen, which are then compiled into the GUI modules in applications. Various CAD systems also interpret drawings, and these systems are used for mechanical simulations. Some CASE tools can recognize UML diagrams and convert them into textual programs. However, they usually do not utilize spatial parser generators. There are several reasons for this. First, the spatial parser generators available at present are not very popular. In addition, it is often difficult to understand complex figures in real applications.

Another type of built-in spatial parser can interpret freehand drawings. For example, the Electronic Cocktail Napkin project by Mark Gross[81] involved development of a pen-based system that can recognize hand-drawn sketches and diagrams. The ultimate aim of this project is the development of a pen and intelligent paper that can support the sketching of designs on the screen. Teddy[57] is another example that enables freehand drawing of three-dimensional polygon models. By only sketching outlines of models, users can create 3D polygon models, which can be used in the design of objects such as the stuffed toys after which it is named.

2.5 GUI Construction Tools

Writing a graphical user interface (GUI) used to be a complicated process, but a great deal of work has gone into reducing the cost of making a GUI. Various toolkits, libraries, and environments are now available for constructing GUI applications. However, most still provide only a basic set of widgets and primitive graphics libraries. This section summarizes the work that goes into making GUI applications, focusing on programming by example, programming by demonstration, and interface builders.

2.5.1 Programming by Example and Programming by Demonstration Approaches

Programming by example (PBE) and programming by demonstration (PBD) are techniques by which the user provides examples or demonstrations to the system, from which the system constructs ”programs” for a given task. For example, EAGER[31] watches the user's operations in a GUI environment, and detects repeated patterns. When a repeated sequence of operations is detected, EAGER asks the user whether it should complete and repeat the sequence. This can be used as a “keyboard macro” in GUI environments.

These techniques have been applied to the construction of GUI applications. Peridot[90] is an early PBE-based user interface construction tool in which the user places graphical objects on a screen and can make them react to input. Peridot infers two types of constraints; i.e., *graphical constraints* that hold among graphical objects placed by the user, such as a constraint that the sizes of rectangles must remain the same, and *data constraints* that hold among graphical objects and the arguments of a function that the programmer is to construct. A *virtual mouse* is used to simulate the user's mouse input. DEMO, DEMOII[42], and Pavlov [128] are also based on PBD. The DEMO system has been used to make a rotating gauge that works together with a number panel, DEMO-II

to make xeyes-like applications by demonstration, and Pavlov to specify the user's control module in a car racing game.

2.5.2 Interface Builders

Most current software development environments, usually called Integrated Development Environments (IDEs), such as Microsoft's Visual Basic or Visual Studio, and Borland's JBuilder, have an *interface builder* that enables programmers to layout widgets interactively in a window and change their various properties such as colors and fonts. Some IDEs allow the programmer to specify callback functions or event handler methods for widgets. This specified information is compiled and linked with other programs to build applications.

The NextStep Development Kit [95] is a well-known early IDE with an interface builder. The user can not only lay out the widgets interactively on the screen, but can also connect a widget to an object via a line that specifies the flow of events at the widgets. In this environment, the user can immediately try the interface without compiling the application.

SILK[60] is a user interface prototyping tool in which the programmer roughly draws a sketch of the user interface with a mouse or a pen. The drawn sketches are recognized by SILK, and the corresponding widgets are arranged automatically. In addition, the user can operate the rough sketch of the interface, i.e., the user can move the drawn scrollbar or push the sketched buttons.

Chapter 3

The Bi-Directional Translation Model

This chapter describes two models: the basic model of visualization and parsing pictorial data, and the extended model for animation. These models form the basis of this thesis, and have been applied to the systems described in the following chapters. Note that , as described in Chapter 1, our model is extended from the framework of TRIP[67] which only handles visualization of abstract data. We integrated parsing pictorial data and animations into the framework.

3.1 Overview of the Bi-Directional Translation Model

The objective of this model is to depict the standard structure of implementations that provide two functions: visualization of application data and interpretation of figures. They are important functions when building direct manipulation interfaces, but are not fully supported in most GUI construction tools. This model aids in building direct manipulation interfaces whose domain is abstract data structures such as trees and graphs.

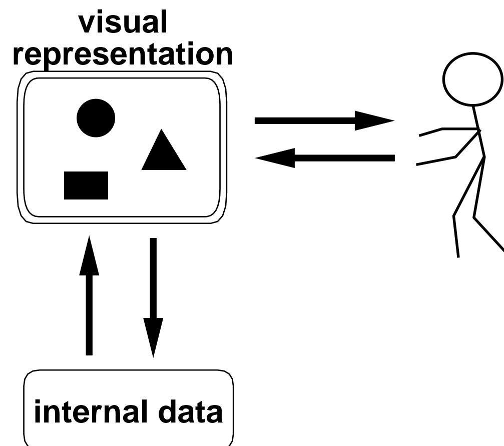


Figure 3.1: The idea of direct manipulation interfaces.

Figure 3.1 outlines the idea of direct manipulation interfaces for the data visually represented on the screen. The diagram on the screen visually represent internal data of the system. The user recognizes the diagram as internal data, and operates on the diagram. These operations on the diagram are interpreted as operations on internal data. That is, the system *visualizes* application data to the diagram on the screen, and *interprets* the diagram on the screen to application data. In our model,

visualization is thought of as translation from source data representation into target pictorial data representation, and interpreting of pictorial data is thought of as the converse, i.e., translation from pictorial data representation into target data representation. To build direct manipulation interfaces, these two functions, visualization of internal data and interpretation of operations on the visual representation, must be implemented. Thus, two translators are required that can perform translations between pictures and internal application data.

Usually, programmers must use canned translators or write such translators from scratch, but such translators resemble each other in their translation process. Our model decomposes the bi-directional translation process into several sub-translations by incorporating intermediate data representations. It enables systematic implementation of bi-directional translations for direct manipulation interfaces. In fact, we have enabled such translations by a set of declarative rules, designated as a *visual (and inverse visual) mapping rule set*. To visualize abstract data, programmers have only to write a visual mapping rule set that maps the ASR of the original abstract data into the VSR of the target picture. In the same way, to interpret visualized pictures, the programmer has only to write an inverse visual mapping rule set. This topic is described in more detail in the following sections.

3.2 Data Representations

There are many types of both data representations to be visualized, and of target pictorial data representations. As the translation process depends on these source and target data representations, it is difficult to reuse the translators if translations are performed between these individual representations. Therefore, two intermediate representations are incorporated into the process of translations; i.e., abstract structure representation (**ASR**), and visual structure representation (**VSR**). ASR consists of a set of ground compound terms. ASR represents the structure of abstract data, and is used as a proxy for the application data in the model. VSR is a high-level representation of a picture, and consists of a set of graphical objects and graphical constraints. By using these two intermediate representations, programmers can specify the mapping between abstract data and a picture independently of each specific representation.

With these two intermediate data representations, our models has four data representations as follows:

Application Data Representation (AR) Any application data can be the source for visualization. For example, data may be the result of a simulation, internal data generated during some computation, or logs of communications between processes. However, their primary purpose is not necessarily visualization. In fact, they usually include data that are not used for visualization or may not have a form appropriate for visualization. To visualize such data, necessary information must be extracted and transformed appropriately for visualization. Similarly, the data obtained by interpreting pictures must be converted for application data representation before they are passed to the application.

Abstract Structure Representation (ASR) Abstract structure representation (ASR) is the user-defined data representation for representing application data in our model. In contrast to application data representation described above, the purpose of ASR is to represent direct data corresponding to the target picture. Therefore, ASRs should be designed so that they do not include data that are unnecessary for visualization, and that they are structurally similar to the corresponding picture.

ASR data are essentially a set of facts. The syntax of facts is the same as that used in Prolog,

i.e.,

$$predicate.$$

or

$$predicate(arg_1, arg_2, ..., arg_n).$$

where arg_1, ..., arg_n are symbols, numbers, lists, or terms. Variables cannot be used as arguments. This syntax is primarily intended to represent symbolic structural and relational data, because, although this model is also applicable to other types of data such as numerical data, we mainly handle abstract structural and relational data such as hierarchical tree structures and network graph structures.

The user determines what kinds of facts are required to represent application data as ASR. For example, if the application has network data, a graph structure representation should be used as the ASR. A simple graph structure can be represented by two types of facts; `node/1`[1] and `edge/2`. For example, a complete graph with three nodes (`a`, `b` and `c`) can be represented as follows:

```
node(a).
node(b).
node(c).
edge(a, b).
edge(b, c).
edge(c, a).
```

For real applications, the definition of the ASR will be more complex. For example, it may have more types of nodes and edges, and nodes and edges may have more attributes to represent more data in the application. Therefore, the corresponding picture can be more complex.

Visual Structure Representation (VSR) Visual structure representation (VSR) is a high-level representation of pictures. It is high-level in the sense that it describes pictures with graphical objects and graphical relations among graphical objects. The VSR does not contain absolute coordinate values, but represents the structures of pictures directly. That is, graphical objects in VSR have their size information, but do not have positional information. Graphical relations represent relative positional relations among graphical objects, but do not specify their absolute positions. For example, the *horizontal* relation put the graphical objects horizontal. However, it does not specify each graphical object's x- and y-coordinate. This is because such graphical relationships are expected to be closely related to the meaning of the picture that represents an abstract relational structure, and thus the mappings between ASR and VSR become simple.

Graphical objects, which represent primitive shapes in a picture, are written as follows:

$$type(id, arg_1, arg_2, ..., arg_n, options).$$

The `type` describes the type of the graphical object, e.g., line, circle, or rectangle. Arguments describe the attributes of graphical objects, such as width and height. The number of arguments differs among the different types of graphical objects. Each graphical object has a unique identifier(ID) that is used to distinguish it from other graphical objects.

Graphical relations are written as follows:

$$type([id_1, id_2, ..., id_m], arg_1, arg_2, ..., arg_n, options).$$

[1]The syntax $predicate/N$ represents a predicate with N arguments. For example, `node/1` is a predicate named `node` with one argument.

Graphical relations also have a type and several attributes. Their first argument is a list of the IDs of graphical objects related to them. Other arguments are specific to each type of relation. Note that graphical relations do not have their own ID. Graphical relations that have the same type and attributes are treated as one relation.

The following is an example of VSR data, indicating two rectangles that are horizontally aligned:

```
rectangle(r1, 30, 30, []).
rectangle(r2, 30, 30, []).
horizontal([r1, r2], []).
x_order([r1, r2], 10, []).
```

The first and second lines are graphical objects with the IDs `r1` and `r2`. Atoms, ground terms, and lists can be used as IDs of graphical objects. The third and fourth lines are graphical relations that indicate that `r1` and `r2` are arranged horizontally and that the distance between the x-coordinates of `r1` and `r2` is 10. As shown in the above example, they have an empty list of options as the last argument. The lists of options contain attributes such as colors of shapes and the method of alignment for geometric constraints. An empty list means that they use default values as attributes.

The available set of graphical objects and relations varies depending on the implementation of the geometric constraint solver in the system. For example, TRIP systems use the COOL system[67]. It has a linear equation solver and an undirected graph layout module, which are convenient for specifying the layout of hierarchical tree structures and network graph structures. The details of VSR implemented in our systems are described in the following chapters.

Picture Data Representations (PR) Any pictorial data representation that is designed to display or print, such as various bitmap representations, saved files for drawing editors, PostScript data, or VRML, can be a target of visualization and a source for spatial parsing. Pictorial data representations are presented directly to the user, and the user interacts with displayed data. Thus, they should have concrete information regarding the coordinates of objects in the picture, and should be ready to be displayed. Conversely, differently from VSR, their structures may not be obvious. It is often necessary to devise a way to parse such representations and extract their structures.

3.3 Various Functions in the Model

Several functions related to visualization of abstract data and interpretation of pictures can be regarded as a combination of translations among the data representations described in the previous section. Figure 3.2 shows an example translation process in the model. Using this figure, this section introduces the interpretations of several functions in this model.

Visualizing Application Data Visualization is a process of generating pictorial representations from source information. This process is modeled as shown by steps (1), (2), and (3) in Figure 3.2, which is same as the model of TRIP[67]. The first step of visualization is the extraction of ASR data from application data. The purpose of this translation is first to convert the syntax of data into the format of ASR, in order to prepare for the next translation process. Second, as described in the previous section, the application data are usually not prepared for visualization. Data that are unnecessary for visualization are often included, and data that are necessary may be distributed over several parts of the application data. This step extracts and aggregates necessary information from application data, and converts it into ASR. In Figure 3.2, translation (1) corresponds to this step. Here, two sentences, "`X consists of p, q, and r`." and "`X, p, q, and r`

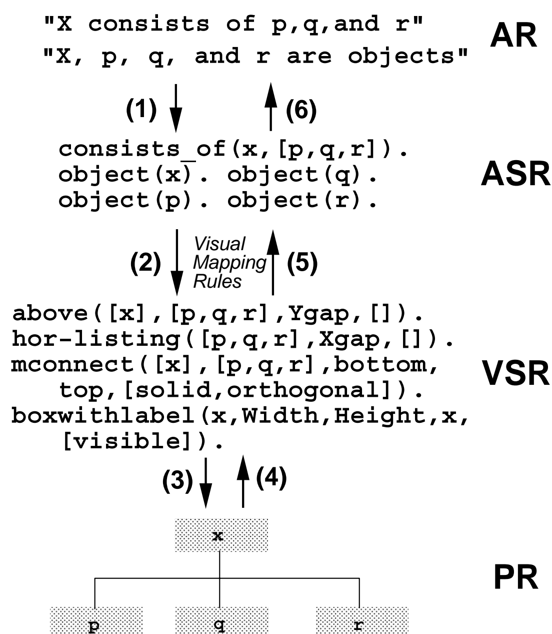


Figure 3.2: A translation example in the bi-directional translation model.

are objects," are translated into five terms. The term "consists_of(x,[p,q,r])" implies that there is a "consists_of" relationship between x and the list p, q, and r. The other four terms imply that x, p, q, and r are "objects". With this translation step, the structure of the application data has become more explicit. In other words, the programmer has extracted the structure to be visualized from the application data.

The second step is the translation from ASR into VSR, which we call *visual mapping*. In this translation, the visual characteristics of the target picture are determined according to visual mapping rules. By changing visual mapping rules, various pictures can be generated from the same ASR data. However, as the picture that we wish to generate by visualizing abstract data is a representation of abstract data from which the user can recognize its structure and meaning, there should be some correspondence between the structure of abstract data and the structure of the visualized picture. The programmer should specify visual mapping rules with this point in view. In addition, if they are structurally similar, visual mapping rules are expected to be simple. Abstract objects and relations in ASR are simply translated into graphical objects and relations in VSR. In addition, the expressiveness of VSR, that is, how many and strong graphical relations are supported by the implementation of VSR is related to the simpleness of visual mapping rules. If enough expressive graphical relations are provided in VSR that are convenient to depict abstract relations, visual mapping rules will be simple. If not, the programmer must combine complex constraints to specify the intended graphical relations.

In Figure 3.2, each object is mapped to a graphical object. For example, object(p) is mapped to

```
boxwithlabel(p, Width, Height, p, [visible])
```

This is a graphical object representing a box labeled with a string whose ID is p. The relation consists_of(x,[p,q,r]) is mapped to three graphical relations:

```
above([x],[p,q,r],Ygap,[])
```

```
hor-listing([p,q,r]),Xgap,[])
mconnect([x],[p,q,r],bottom,top,[solid,orthogonal])}
```

These relations lay out the objects so that (1) the object `x` is placed above the other objects, `p`, `q`, and `r`, (2) objects `p`, `q`, and `r` are arranged horizontally, and (3) object `x` is connected by a line to the other objects, `p`, `q`, and `r`. These mappings are specified by the visual mapping rules written by the programmer. Figure 3.3 shows the visual mapping rules used for the mappings in Figure 3.2. Each rule maps an ASR term on the left-hand side of a rule to VSR terms on the right-hand side. The syntax of rules varies depending on the implementation. The rules shown in Figure 3.3 are those for the TRIP2 system described in Chapter 4.

```
object(X) :-
     boxwithlabel(X,Width,Height,X,[visible]).
consists_of(X,L) :-
     above([X],L,Ygap,[]),
     hor-listing(L,Xgap,[]),
     mconnect([X],L,bottom,top,[solid,orthogonal]).
```

Figure 3.3: An example of visual mapping rules.

In the last step of visualization, PR data — a picture in the form of a target representation is generated from VSR. VSR data are a set of high-level graphical objects and geometric relations, and do not have absolute positional information. The layout of the picture are specified by the graphical relations among graphical objects in VSR. In order to generate PR data, positional information of the picture must be obtained. Therefore, in this step of translating VSR into PR, first, absolute coordinates of the elements in a picture are calculated by solving geometric constraints of graphical relations in VSR data. Second, target picture data are generated by substituting the calculated results for the coordinate variables of graphical objects. Although this step depends on the representation of target picture data, the module that solves constraints of graphical relations in VSR can be used for all translators for this step. In Figure 3.2, this step is shown as translation (3). The result of translation is the tree shown at the bottom of Figure 3.2.

In summary, visualization of abstract data is a process of translating AR into PR via ASR and VSR. It is a process of extracting the structure of the data to be visualized from the application data, mapping abstract objects and relations into graphical objects and relations, and calculating the layout of elements in the picture according to the corresponding graphical relations in VSR.

Interpretation of a Picture A figure that represents abstract data is a proxy for that data. Thus, it can be used not only for viewing data, but also for handling abstract data. People draw and modify a figure when they want to create or change abstract data. Thus, we must provide not only a means to visualize abstract data, but also a means to reflect the changes to application data. In our model, this function is regarded as an inverse translation from pictorial data to application data (Figure 3.2, steps (4), (5), and (6)).

First, pictorial data are translated into VSR data, a process referred to as *spatial parsing* (Figure 3.2(4)). In this translation, geometric objects and relations in the picture are recognized by parsing the picture, and output in the format of VSR. The translator for this step must be constructed for each pictorial representation. The purpose of this translation is similar to the translation from AR to ASR. That is, the translation converts various PR syntax into the format of VSR in preparation for the next translation. In addition, the structure of pictorial data is not usually obvious. In

this translation from PR into VSR, the important graphical relations in the picture that related to its 'meaning' is extracted from the picture, and explicitly represented as VSR data. Note that there are often multiple ways to interpret pictures, and there are so many graphical relations unrelated to the interpretation of the picture. This translation step extracts only meaningful graphical relations from a picture in reference to the next translation step.

The second step of the inverse translation is the translation from VSR into ASR, which we call *inverse visual mapping* (Figure 3.2(5)). In this step, the "meaning" of visual information is determined; i.e., graphical objects and graphical relations in VSR are mapped to abstract objects and abstract relations in ASR. As this translation is the inverse of the translation from ASR to VSR, the visual mapping rule set that maps ASR into VSR can be applied inversely for this inverse translation. However, for translation between abstract data and pictures, there are some differences between visual mapping and inverse visual mapping. Visual mapping *adds* geometric information to abstract data in order to visualize them, because abstract data do not have their own shape or layout. On the other hand, inverse visual mapping must *eliminate* layout information, and extract only the structure of the picture. Therefore, mapping rules are applied differently in each direction of translation between ASR and VSR. When mapping inversely from VSR to ASR, some geometric constraints in visual mapping rules are often ignored. The system tolerates some error of geometric constraints. For example, when visualizing a node in a graph as a box, it is necessary to determine its size precisely with visual mapping rules. However, when recognizing a box as a node, all boxes of approximately the same size should be mapped to nodes to deal with pictures drawn not so precisely by the user. In Figure 3.2, the graphical object

```
boxwithlabel(x, Width, Height, x, [visible])
```

is translated into the object `object(x)` in ASR. The three graphical relations:

```
above([x],[p,q,r],Ygap,[])
hor-listing([p,q,r]),Xgap,[])
mconnect([x],[p,q,r],bottom,top,[solid,orthogonal])}
```

are translated into the abstract relation `consists_of(x,[p,q,r])` in ASR.

In the last step of inverse translation (Figure 3.2(6)), ASR data are translated to application data. The purpose of this step is mainly conversion of the data representation. ASR data are converted to various types of application data, such as data structures in C/C++ and S-expressions in LISP. The converted data are passed to the application.

In summary, interpretation of a picture is a process of translating PR into AR via VSR and ASR. It is a process of translations that extract important graphical relations from pictures, map graphical relations and objects in VSR into abstract relations and objects in ASR, and compose the application data from the ASR data.

Direct Manipulation of Abstract Structure Data Using the bi-directional translation between application data (AR) and picture data (PR) as described above, a kind of direct manipulation interface, especially direct manipulation user interfaces for handling relational structure data, can be interpreted in our model.

As an example, here we consider the construction of a user interface for the application that handles entity-relationship (ER) diagrams as input. In this application, an ER diagram is displayed on the screen. The user draws a diagram, or modifies the diagram displayed by the system. The application has the schema data, upon which it computes, internally. The schema data are visualized and displayed to the user, and the user manipulates the visualized diagram. (Figure 3.4).

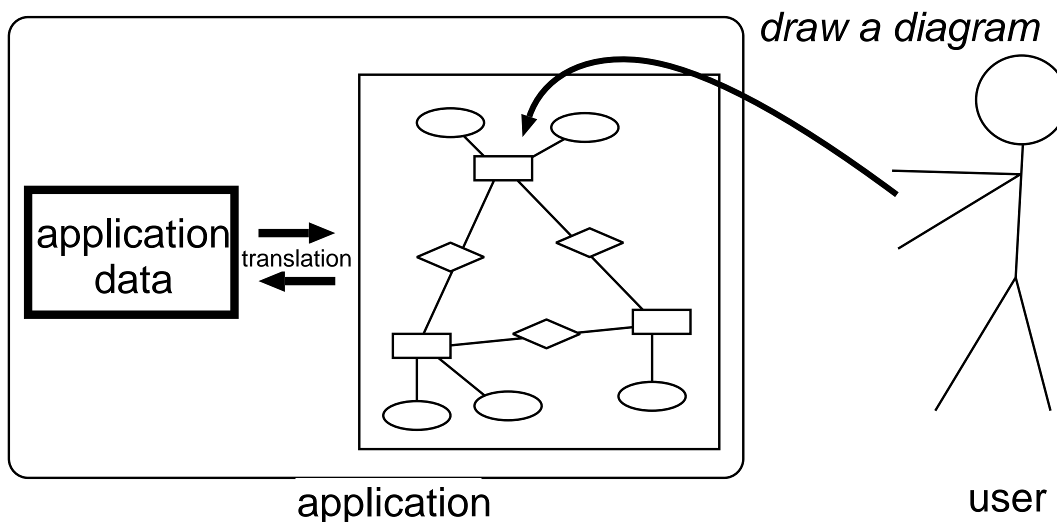


Figure 3.4: Architecture of ER Diagram Editor Application.

To construct the user interface part of this application, the following modules have to be implemented:

1. A module that draws a diagram that reflect the state of the internal data of the application.

2. A module that enables the user to draw and modify the diagram on the screen.

3. A module that recognizes the user's input, and sends it to the application.

The function of the first module is visualization, so that, in our model, it can be thought of as performing the translation from application data to pictorial data. On the other hand, the function of the third module is interpretation of a picture. That is, it corresponds to translation from pictorial data to application data. Therefore, these modules can be realized by bi-directional translations between application data and pictorial data, which are specified by visual mapping rules. The function of the second module is to allow users to draw pictures as the input to the third module. A general drawing editor module, such as Adobe Illustrator[2] and TGIF[3], is sufficient for this purpose, and can be shared among various applications.

In general, direct manipulation-style (DM-style) user interfaces can be modeled as two parts; the bi-directional translator between application data and pictures, and an interaction module for manipulating visualized pictures. TRIP2 is a system that aids in building these parts. With TRIP2, only a set of visual mapping rules is necessary to build DM-style user interfaces. Chapter 4 describes the TRIP2 system in more detail.

In summary, in our model, direct manipulation is modeled as a bi-directional translation between application data and pictorial data. Application data (AR) and picture data (PR) have extra information. ASR data and VSR data are extracted from them to represent the important structure of the data required for the bi-directional translation. With these intermediate data representations, the programmer can specify visual mapping rules that define the mapping between ASR and VSR.

[2] A powerful vector graphics tool developed by Adobe Systems Incorporated.
(http://www.adobe.com/products/illustrator/main.html)

[3] An interactive 2-D drawing tool under X11 on various UNIX platforms.
(http://bourbon.usc.edu:8001/tgif/index.html)

The translation between ASR and VSR is the essential translation in the sense that it couples the abstract concept and the visual representation. Note that the other translations — translation between AR and ASR and translation between PR and VSR — are also important for completing the whole translation. They should be also supported in the development environment of the bi-directional translations. The former is related to the application interface of the user interface part developed by the system. The latter is the problem of geometric constraint solver and spatial parser of the development system. Their strength and usability have influence on the cost of developing direct manipulation interfaces.

3.4 Extended Model for Animation

Animating application behavior is important and useful for various areas, such as education, debugging, and performance tuning. In this section, we extend the basic model described in the previous sections to accommodate the creation of animations in the model.

Basic Idea Application programs compute on their internal state during their execution. That is, execution of an application is a sequence of operations and states. When an operation is executed, the application's state is changed, and goes to the next state in that sequence. Animations can also be viewed as a sequence of pictures. If the target of an animation is a program execution, each picture in the animation represents the application's internal state, and the picture changes represent operations in the execution.

We have extended the bi-directional translation model for the creation of animations as shown in Figure 3.5. The model is composed of a series of data for each representation. In addition, four types of "operations" are incorporated into each representation in the model. In Figure 3.5, the vertical sequence $\langle AR(t) \rightarrow ASR(t) \rightarrow VSR(t) \rightarrow PR(t)\rangle (0 \leq t \leq N)$ corresponds to the process of translating application data at time `t` to picture data, as described in the previous section. The time `t` is discrete, and starts from 0 when the application is first invoked. It is incremented when an operation is executed in the application. The time is `N` when the application is terminated.

On the other hand, the four horizontal sequences in Figure 3.5 represent changing data and operations on each representation. For example, the horizontal sequence at the top of Figure 3.5, $\langle ... \rightarrow AR(t) \rightarrow AR(t+1) \rightarrow AR(t+2) \rightarrow ...\rangle$, is a series of application data and operations. At time `t`, an operation is executed and the state is changed from $AR(t)$ to $AR(t+1)$. Then, another operation is executed and the state is changed to $AR(t+2)$. This sequence continues until the application is terminated. The other three horizontal sequences are similar to the sequence of $AR(t)$. In the sequence of $XR(t)$, an operation on XR changes $XR(t)$ to the next state $XR(t+1)$ where $XR = AR$, ASR, VSR, or PR.

Specifying Transitions In the extended model, an animation corresponds to the horizontal sequence of $PR(t)$ and operations on PR. In fact, an operation on PR is the short animation $A(t)$, the steps of which are connected to form a whole animation A; i.e., $A = A(0) + A(1) + ... + A(N-1)$. The short animation $A(t)$ is generated by interpolating two successive pictures $PR(t)$ and $PR(t+1)$. Each picture $PR(t)$ can be generated by translating application data $AR(t)$.

In summary, using the extended model, we can create an animation by (a) obtaining the application's internal data during its execution, (b) translating the sequence of application data to a sequence of pictures, and (c) generating a sequence of short animations by interpolating the sequence of visualized pictures. As the translation from application data to a picture (i.e., (b)) is achieved only by specifying visual mapping rules, as described in Section 3.3, it is possible to specify an animation

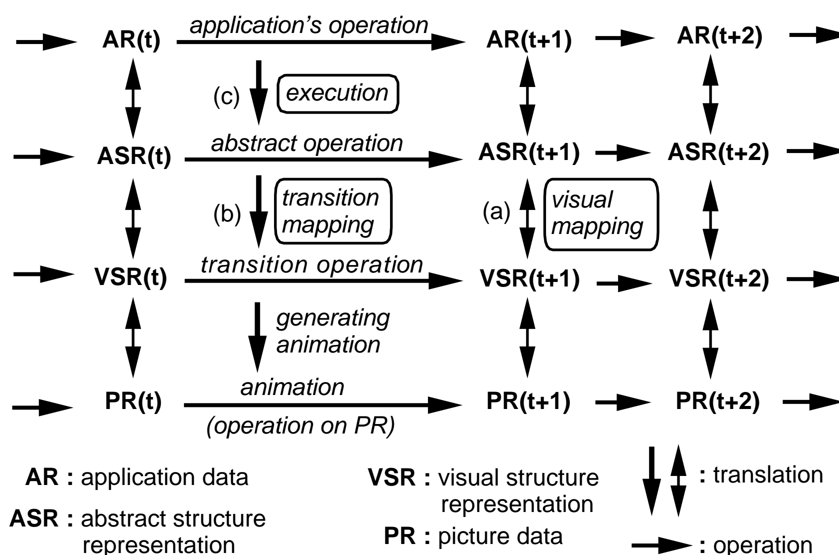


Figure 3.5: The extended bi-directional translation model that represents the general architecture of applications that generates animations of program execution.

using only visual mapping rules. However, since there are many possible ways to interpolate two pictures, it would be better to provide a method for their specification.

In our model, the method of interpolating pictures is regarded as a *transition operation* $to(t)$, which is an operation on VSR that changes $VSR(t)$ to $VSR(t+1)$. This is a method of moving, scaling, and rotating the objects in one picture to those in another picture. In addition, a transition operation is translated from an application's operation via an *abstract operation* $ao(t)$ on ASR, which is defined as an operation that changes $ASR(t)$ into $ASR(t+1)$. The translation from an abstract operation to a transition operation is specified by mapping rules, designated as *transition mapping rules*.

The process of translating application operations to an animation is illustrated in Figure 3.5. First, the application executes an operation; in our model, it is an operation on AR. Then, an abstract operation defined on ASR is invoked. As it is almost impossible to automatically recognize meaningful operations for animations from an application program, programmers themselves must define abstract operations; that is, they must insert code into the program to define abstract operations. Abstract operations correspond to *interesting events* in BALSA-II[15] or Zeus[16].

Then, abstract operations are translated to transition operations by transition mapping. As a transition operation is specified for one graphical object in VSR, some graphical objects in VSR are associated with transition operations specified by the mapping rules. Other objects are associated with default transition operations. Finally, the system generates a short animation by inbetweening the previous and the new picture according to the transition operations.

For example, suppose that the programmer is to make an animation that depicts the insertion sort algorithm. It shows the process of sorting bars of different lengths in ascending order from left to right. The insertion sort algorithm sorts bars by inserting a new bar into a row of bars already sorted. Figure 3.6 is an example animation that a bar is inserted at the appropriate place of the sorted row of bars. To move a bar in a circular motion, as in Figure 3.6, the following transition mapping rule is specified:

```
insert(X) :- move(n(X), [clockwise]).
```

This mapping rule maps the operation `insert(X)` on ASR to the operation

move(X, [clockwise]) on VSR. Here, insert(X) represents an operation in the sorting program. X is a string or integer that represents the index corresponding to the inserted object in ASR, and n(X) is the name of the inserted graphical object in VSR[4]. The transition operation move(n(X), [clockwise]) specifies the method for moving the graphical object n(X) as clockwise. Thus, when insert(X) is executed, the graphical object named n(X) moves clockwise. The system moves the rest of the graphical objects, which are not governed by any transition operations, using the default transition operations. The default transition operation for moving a graphical object is move(Obj, [straight]), which moves the graphical object Obj in a straight line. In Figure 3.6, two small bars are shown moving straight to the left to make room for the bar being inserted.

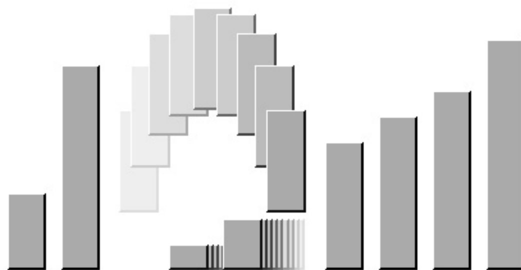


Figure 3.6: Circular movement of a bar in a sorting animation.

[4]The name of a graphical object is specified in a visual mapping rule.

Chapter 4

TRIP2 — A System For Constructing Direct Manipulation Interfaces

4.1 Introduction

Graphical and interactive user interfaces based on *direct manipulation*(DM)[109], allowing the users to point to, grasp, and drag objects on the screen, are now standard. To ease the high creation cost, we employ User Interface Management Systems (UIMSs) and user interface toolkits such as NextStep[95], MacApp[3], InterViews[113], GTK+[98], and Java Swing set[125]. However DM style interfaces are still costly, because UIMSs and toolkits usually only provide 'canned' abstractions of behaviors for a fixed set of simple graphical interface objects such as buttons and scrollbars. Such a fixed set of abstractions may be sufficient for simple, dialogue-style interfaces, but for cases where the semantics of complex application data must be reflected and be manipulatable on the display, appropriate graphical and input abstractions may not be available. This is primarily because current models of user interfaces are usually the models of the interaction architecture, and lack the support for consistent framework that allow visualization and manipulation of high-level abstract data, i.e., semantics of applications.

In this chapter, we show an implementation of a framework for bi-directional translation between application's data representations and pictorial representations of the user interface. With this framework, the feedback of user's manipulation can be generally achieved via inverse translation from picture data into abstract data. The specification of inverse translation is achieved with little extra cost, because (almost) identical rules can be used for mappings in both directions.

We have implemented TRIP2, a prototype system based on this model. In TRIP2, direct manipulation interfaces for various types of data can be achieved just by specifying declarative mapping rules.

4.2 The TRIP2 System

TRIP2 is a prototype system based on our bi-directional translation model, which is implemented on the NeXT computer. This section describes the implementation of TRIP2.

4.2.1 System Overview

Figure 4.1 shows an overview of the TRIP2 system. TRIP2 evolved from TRIP (TRanslation Into Pictures)[65, 67], which handled only one-way translation from abstract data into picture data. The

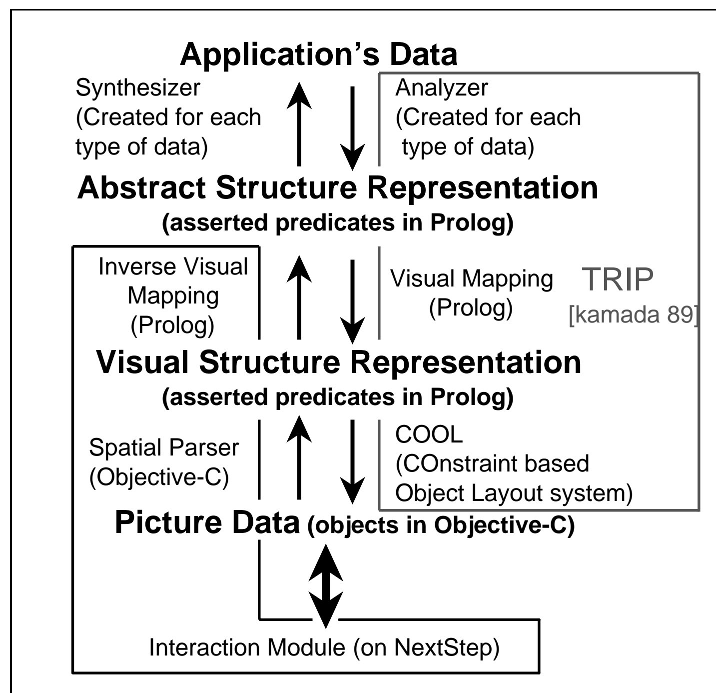


Figure 4.1: The architecture of the TRIP2 system.

new features of TRIP2 that realize our bi-directional translation model are as follows:

1. The **interaction module** which allows the users to manipulate the pictures directly with input devices such as a mouse. Currently, this module provides the users with an object-oriented freehand drawing interface similar to MacDraw[1]. This module is implemented on NextStep[2] by using Objective-C; thus, low-level graphical interactions, such as selection of a picture object with a mouse, are treated as method invocation of Objective-C objects that represent the picture.

2. The **spatial parser** which translates picture data into its visual structure representation. In TRIP2, picture objects are mapped directly to graphical objects in its visual structure representation, and this spatial parser extracts graphical relations from the picture data.

3. The **inverse visual mapping module** that translates visual structure representation into abstract structure representation. This translation is specified by inverse mapping rules given as a set of Prolog predicates.

In TRIP2, since the visual mapping and the inverse visual mapping are executed in Prolog[3], we are using Prolog's asserted predicates for ASR and VSR data. The spatial parser is integrated with the interaction module in the current implementation. The details of these modules are described in the following sections.

The users interact with TRIP2 in the following way:

1. Compile the mapping rules, and read them into the rule database.

[1]MacDraw is a trademark of Claris, Inc.
[2]NextStep is the operating system developed by NeXT Computer.
[3]SB-Prolog version 3.1, Copyright (C) 1986 at Stony Brook; 1987 University of Arizona.

2. Read the ASR data from a file into the *data* window.

3. Execute TRIP to translate the data into a picture.

4. Edit the picture in an object-oriented freehand manner.

5. Execute the *flush* command. This command first translates VSR data into ASR data, and then re-executes TRIP, i.e., translates the new ASR data into a picture. As a consequence, application semantics-directed beautification is achieved.

6. Return to 4.

Here we show an example in order to give some guidelines for using TRIP2 in Figure 4.2. Each screen dump of left-right windows shows the data of a Japanese family as ASR and its corresponding picture as a kinship diagram (Figure 4.2(a)). The left window is the *Data Window* which displays ASR data, and the right window is the *ObjView Window* which displays its pictorial representation. The user is free to modify either data at any time. (He interacts only with the pictures in the *ObjView Window* in this example.) Now, suppose that a new member, *Midori*, joins the family by marrying *Shinichi*. To express this fact, we draw a rectangle (Figure 4.2(b)), write the label *Midori*(Figure 4.2(c)), and connect it to the rectangle labeled with *Shinichi* by a line(Figure 4.2(d)). (Note that this label *Midori* was not produced by an explicit command to label the rectangle; rather, it is a textual picture object that has an equal status to the rectangles and lines just drawn.) Then, we execute the *flush* command. These modifications are translated back into ASR data, and the kinship diagram is redrawn from the new data (Figure 4.2(e)). Now a baby, *Tomomi*, is born: We draw a circle between two rectangles, draw a new rectangle, write the name, and connect the circle and the rectangle (Figure 4.2(f)). Finally, we execute the *flush* command again. These modifications are fed back, and the diagram is regenerated (Figure 4.2(g)).

4.2.2 The TRIP Module

The TRIP[65, 67] module is responsible for the translation from abstract data into concrete pictures. As illustrated in Figure 4.1, the TRIP module consists of three submodules (1) application's data analyzer, (2) the visual mapper, and (3) the layout system called COOL (COnstraint-based Object Layout system).

The analyzer translates application's data into ASR data, which is a set of Prolog predicates. In [67], examples of several types of application's data are presented, such as English sentences representing kinship relations, S-expressions in Lisp, and C programs. Those analyzers are not an integral part of TRIP2, but TRIP2 provides an editing window for ASR data as shown in Figure 4.2[4]. Thus, the users can interactively manipulate both ends of application-independent representations at the same time.

ASR data are then translated into VSR data by the visual mapper written in Prolog. ASR data and the mapping rules are read into the database, after which the visual mapper is invoked.

Finally, VSR data are translated into PR data by COOL. In COOL, VSR data are declared as special predicates that generate (a) drawing data, and (b) constraints on the geometric attributes in the drawing data. For example, the predicate `box` generates (a) data for drawing a rectangle, and (b) constraints on coordinate variables of the rectangle. The predicate `horizontal` generates (b) constraints on y-coordinate variables of picture objects corresponding to the graphical objects listed

[4] In fact, the original TRIP system was interfaced with a simple natural language parser; as a result, the user could input sentences describing a scenery, and TRIP would automatically visualize it[67]. This parser can easily be interfaced with TRIP2 as well.

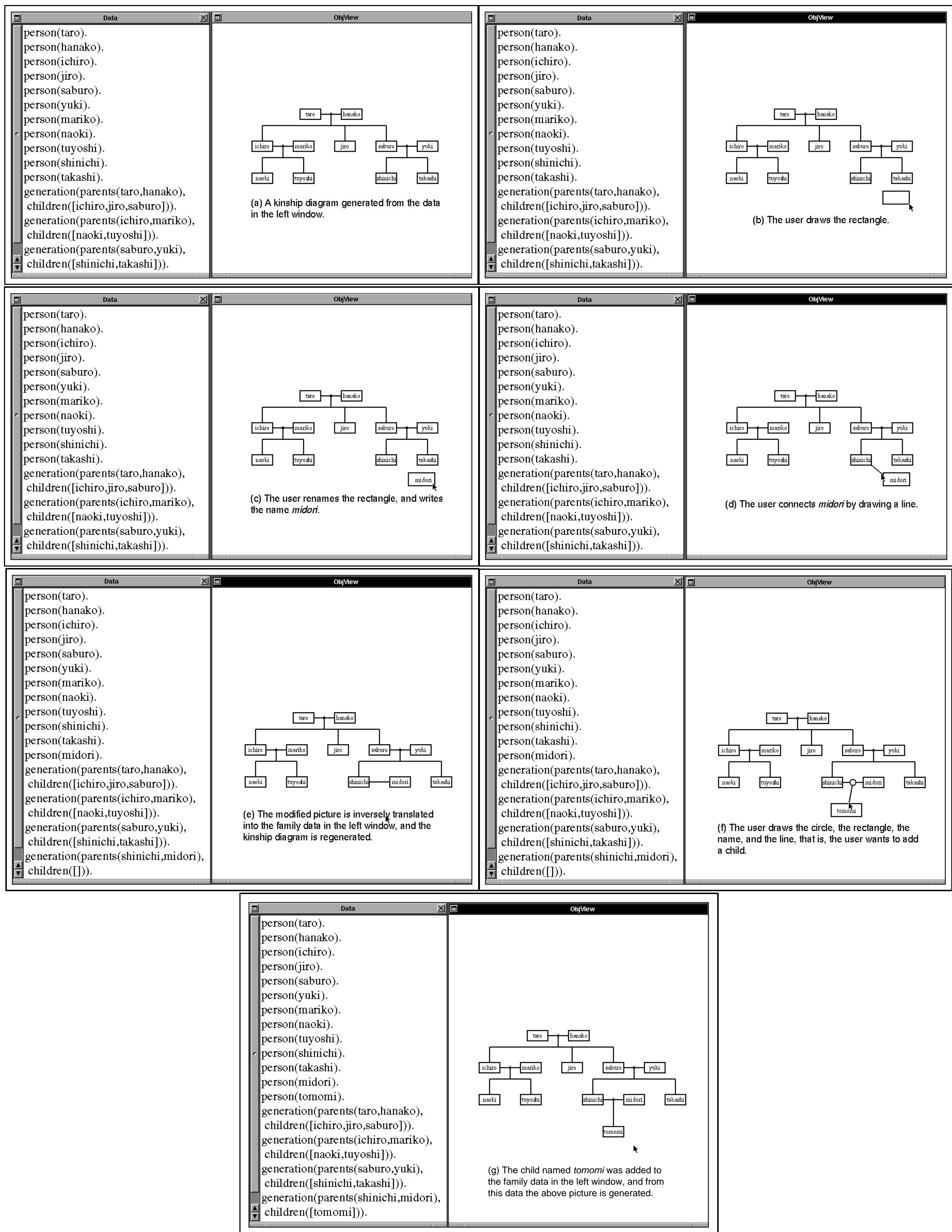


Figure 4.2: A kinship diagram.

in its argument. These constraints are solved by the constraint solvers that are parts of COOL. The constraint solvers can handle two types of constraints: 'rigid' constraints, which must be solved exactly, and 'pliable' constraints, which may be solved approximately by the least square method. The purpose of pliable constraints is to allow COOL to handle over-constrained systems. In addition, COOL has special predicates and another constraint solver for visualizing undirected graph structures as balanced graphs. The constraint solving algorithms of COOL are described in [65, 67]. The solutions computed by the constraint solvers instantiate the coordinate variables in the drawing data. They are then passed to the interaction module to generate the Objective-C picture objects.

For example, given the ASR data:

```
item('AR').
item('ASR').
item('VSR').
item('PR').
order(['AR','ASR','VSR','PR']).
```

and the visual mapping rules (Figure 4.3), the VSR data in Figure 4.4 are generated.

```
objectmap([itemmapping]).
relationmap([ordermapping]).
itemmapping :-
    item(NAME),                       % gets 'item' data from database
    itemmap(NAME),                    % maps 'item' data
    fail.                             % iterates to the next data
itemmap(NAME) :-                      % item(NAME) is mapped to a box with a la-
bel
    box(NAME, 200, 100, []),          % box (width=200, height=100)
    label(l(NAME), NAME, []),         % label 'NAME'.
    contain(NAME, l(NAME), 0, []).    % a box contains a label

ordermapping :-
    order(ILIST),                     % gets 'order' data from database
    ordermap(ILIST).                  % maps 'order' data
ordermap(ILIST) :-                    % items are horizontally arranged.
    horizontallisting(ILIST, 20, []).
```

Figure 4.3: Visual mapping rules example.

To execute the visual mapping, TRIP calls the predicates listed in the argument of predicates `objectmap` and `relationmap`. The predicates of the `objectmap` (`itemmaping`, in Figure 4.3) maps abstract objects to graphical objects, and the predicates of the `relationmap` (`ordermapping`) maps abstract relations to graphical relations. When the predicate `itemmaping` is called by the system (Figure 4.5 (1)), `item(NAME)` gets one ASR data (`item('AR')`) with unification (Figure 4.5 (2)). Then `itemmap(NAME)` unifies the head of the other rule (Figure 4.5 (3)), and the three VSR predicates output VSR data, `box`, `label`, and `contain` (Figure 4.5 (4)). The special predicate `fail` forces backtracking, so this process is repeated with the other `item` until all `item`s are mapped to VSR. Similarly, the predicate `ordermapping` applies the predicate `ordermap` to the abstract relation `order(ILIST)`, and it maps `order(ILIST)` to graphical relation `horizontallisting`. Thus the '`item`'s are layed out horizontally. Next, COOL is invoked and the PR shown in Figure 4.6 is generated from the VSR.

```
% Graphical Objects and Relations mapped from 'item'
box('AR', 200,100,[]). label(l('AR'), 'AR', []).
contain('AR', l('AR'), 0,[]).
box('ASR',200,100,[]). label(l('ASR'),'ASR',[]).
contain('ASR',l('ASR'),0,[]).
box('VSR',200,100,[]). label(l('VSR'),'VSR',[]).
contain('VSR',l('VSR'),0,[]).
box('PR', 200,100,[]). label(l('PR'), 'PR', []).
contain('PR', l('PR'), 0,[]).
% Graphical Relation mapped from 'order'
horizontallisting(['AR','ASR','VSR','PR'], 20, []).
```

Figure 4.4: Generated VSR data.

4.2.3 The Interaction Module

The users of TRIP2 manipulate pictures directly via the interaction module, which provides the generic lowest-level editing facility. They can create, move, and delete picture objects such as rectangles, circles, etc. We stress that the resulting interface is not a structured editor for some specific data — rather, the users simply draw rectangles or circles as if they were using object-oriented freehand drawing systems such as MacDraw. This is unlike graphical editors which are customized for each application, such as Unidraw[124].

A picture in the interaction module is represented as a set of picture objects in Objective-C. Picture objects are the instances of the subclasses of the `Shape` class (see 4.6), in which the usual generic methods, such as drawing, moving, changing the size/place of the object, getting the attributes of the object, and testing whether a given point is within the object, are defined. Several coordinates describing the picture objects, such as the center (`cx`, `cy`), the top left corner (`lx`, `ty`), and the bottom right corner (`rx`, `by`), are declared as instance variables of the `Shape` class. These coordinates are used to test whether geometric relations among the objects hold. For example, when two objects are tested to be horizontal in the *top_align* mode, the *ty* coordinates of these objects are compared.

As mentioned before, VSR data is a set of graphical relations among graphical objects expressed as Prolog predicates. However, since the translation between PR and VSR is executed in the interaction module written in Objective-C, the interaction module must also have the data corresponding to VSR. Therefore, the graphical objects in VSR are mapped idempotently to the picture objects in the interaction module, i.e., the picture objects serve as the dual role of also being the VSR graphical objects. Graphical relations in VSR are represented as a list of Objective-C graphical relation objects, which are the instances of the subclasses of the `Relation` class (see 4.6), which supports the following methods:

- A method for calculating the error of a graphical relation. This method is used for testing whether the graphical relation holds or not.
- A method for testing whether a graphical object is 'addable' to a graphical relation, i.e., testing whether the graphical relation holds when the graphical object is added. This method is used for translating PR data into VSR data.

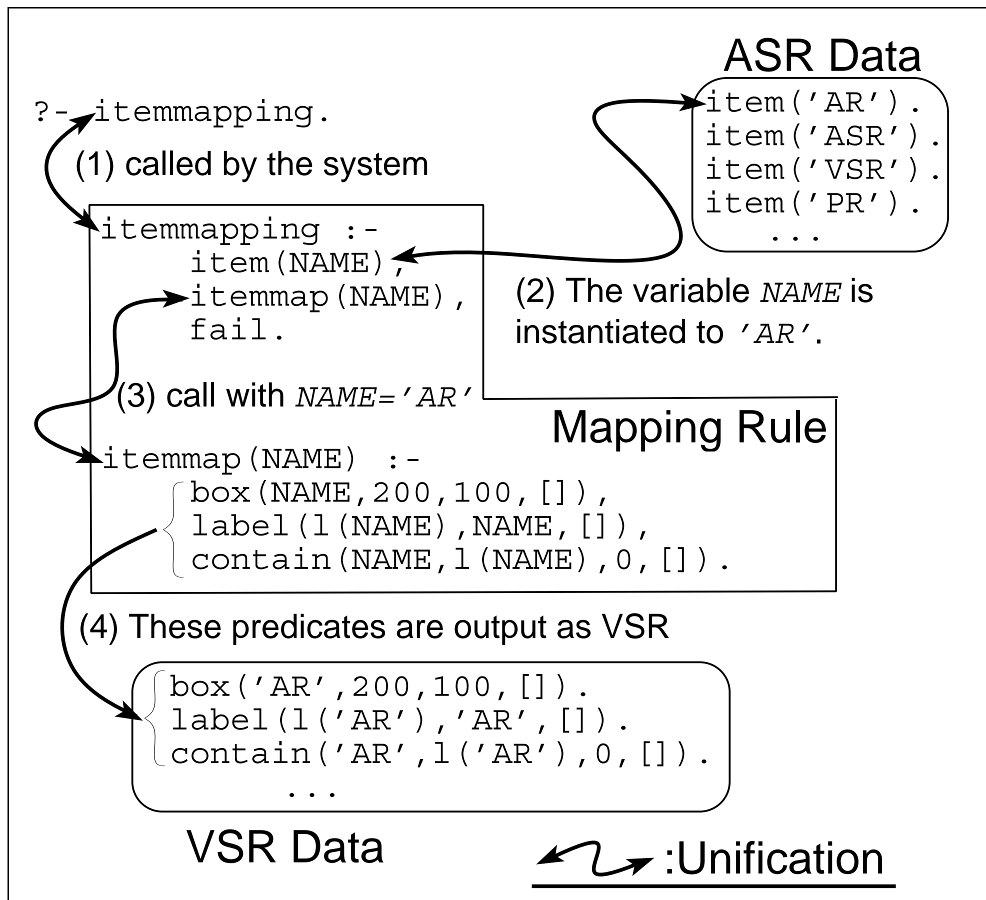


Figure 4.5: The execution of mapping rule.

Figure 4.6: The generated picture.

4.2.4 Implementation of the Inverse Translation

The inverse translation involves translation from PR into VSR, followed by translation from VSR into ASR.

Translation from PR into VSR — Spatial Parsing Here, we refer to translation from PR into VSR as *spatial parsing*. Although spatial parsing has been previously studied (for example, [45, 75, 74]), its application to GUI poses difficulties due to the requirement of real-time use, because there are a great number of graphical relations with no a priori information in a picture.

In our model, spatial parsing has a more restricted role; it extracts a set of graphical relations among graphical objects. Since pictorial objects are recognized as graphical objects in the interaction module, the spatial parser in TRIP2 needs only to extract the graphical relations.

To realize real-time responsiveness, we use an *incremental* spatial parser, which retains most of the graphical relations and updates only its modified portions. Our incremental spatial parser is invoked in the following situations:

- **When a picture object is deleted:** The graphical object (in VSR) corresponding to the deleted picture object (in PR) is removed. In addition, it is removed from each (existing) graphical relation containing it. For example, when the rectangle with a label 'PR' in Figure 4.6 is deleted, 'PR' is removed from the `horizontallisting` relation, that is,

  ```
  horizontallisting(['AR','ASR','VSR','PR'], 20, [])
  ```

  is modified to

  ```
  horizontallisting(['AR','ASR','VSR'], 20, []).
  ```

  To achieve such modifications efficiently, the interaction module maintains a table that maps each graphical object to a list of graphical relations containing it.

- **When a new picture object (`obj`) is created:** The message `addWithTest:obj` is sent to each existing graphical relation object. The invoked method tests whether the `obj` is 'addable' to the graphical relation, and if so, adds it to the graphical relation.

  For example, consider the situation depicted in Figure 4.7. In Figure 4.7, four boxes are constrained by the graphical relation `x_order([a,b,c,d],10)` so that they are placed at even intervals[5]. When a user draws a new rectangle with a label `e` on the right of rectangle `d`, `addWithTest:e` message is sent to the graphical relation object, `x_order([a,b,c,d],10)`. That is, the system checks whether `e` can be added to the graphical relation or not. In this case, the distance `w` between `d` and `e` is nearly equal to 10; as a result, the new rectangle object is added to the `x_order` relation. The spatial parser of our system permits a certain amount of error, because the user usually cannot put new objects at the exact place that satisfy existing graphical relations.

- **When a picture object is moved:** The spatial parser is also invoked when a picture object is moved; this change is treated as a combination of deletion and addition.

[5] It constrains only each box's x-coordinate. Their y-coordinates do not have any meaning in this figure.

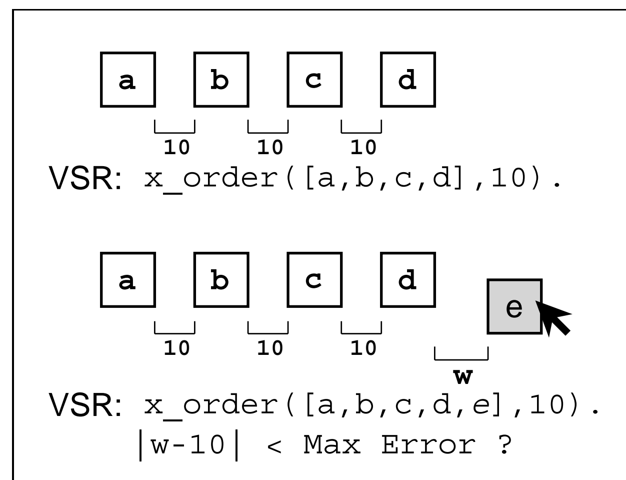


Figure 4.7: Adding a new graphical object to a relation.

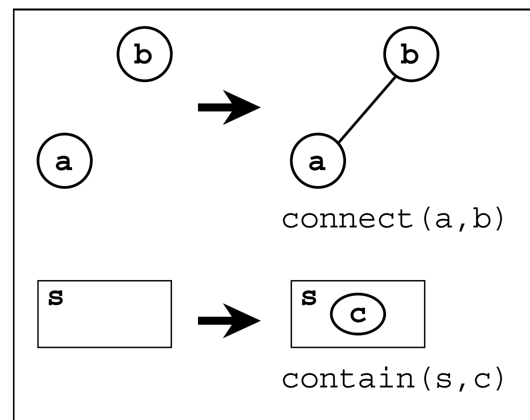


Figure 4.8: Creating a new graphical relation.

- **Adding a new graphical relation:** Although the above scheme can handle the modification of existing graphical relations with an arbitrary number of graphical objects, the shortcoming is that it cannot handle the creation of new graphical relations. For example, the `connect` relation, which represents two graphical objects connected by a line, must be created when a user draws a line that connects two picture objects that were previously unrelated. We alleviate this by associating generations of new graphical relations only with drawings of the picture objects, that is, when a certain kind of an object is drawn, the system checks whether the object has established new relations or not. For example, as shown in Figure 4.8, when a line connecting two objects is drawn, a `connect` relation is created, and when an object is drawn within the boundary of another object, a `contain` relation is created. We are now working to provide alternative means of generating other types of graphical relations.

In TRIP2, all graphical relations are inferred by the spatial parser. One problem of this method is that users cannot see that the system is inferring correctly or not. It is preferable that inferred constraints are displayed visually. In addition, there may be cases that explicit specification of constraints is preferable. These problem has been solved by our next system TRIP3[88].

Translation from VSR into ASR Translation from VSR into ASR (inverse visual mapping) is executed in Prolog. The interaction module outputs VSR data to a Unix pipe, which are read into the Prolog database. VSR data are translated into ASR data according to the inverse mapping rules.

Ideally, we should be able to use identical rules for inverse visual mapping as well as for visual mapping, because declarative rules do not have 'directional' information. However, this is not the case due to the following reason: If some rule is used naively for mappings in both directions, the pictorial editing performed by the user must exactly satisfy the layout constraints imposed by the visual mapping rules. The resulting interface would be extremely rigid and difficult to use — for the example, in Figure 4.3, the user would be required to draw a rectangle of the exact size as specified by the mapping rule ($width = 200, height = 100$) for his input to be recognized.

Instead, we must relax the constraints so that system would be tolerant of the non well-formed operations of the user. In TRIP2, this is currently achieved by having the user provide a separate set of corresponding inverse visual mapping rules which would be 'less specific' compared to its visual mapping counterparts. For example, in Figure 4.9, the term `box` has several of its variables unspecified (this is indicated by underscores ('_'), which serves as *anonymous variables* in Prolog). As a result, the height and the width of the user-drawn box are ignored, allowing the user to draw a box of arbitrary size[6]. Fortunately, the structure of the mapping rules are essentially identical; thus, an experienced user will be able to derive one from the other with ease. Moreover, automatic derivation of inverse mapping rules from the other mapping rules can be possible.

```
invomap([inv_itemmapping]).
invrmap([inv_ordermapping]).
inv_itemmapping :-
    inv_itemmap(NAME),              % finds 'item' data
    asr(item(NAME)),                % outputs 'item' data
    fail.                           % tries for another data

inv_itemmap(NAME) :-                % 'item' data is a
    box(A, _, _, _),                % box with its name 'A'
    label(B, NAME, _),              % and the label 'B' whose string is 'NAME'
    contain(A, B, _, _).            % box 'A' contains label 'B'

inv_ordermapping :-
    inv_ordermap(ILIST),            % finds 'order' data
    asr(order(ILIST)),              % outputs 'order' data

inv_ordermap(ILIST) :-                    % 'order' data is a
    horizontallisting(ILIST, _, _).       % horizontal ordering
```

Figure 4.9: Inverse visual mapping rules example.

For example, the inverse visual mapping rules that correspond to the visual mapping rules in Figure 4.3 are shown in Figure 4.9. They are executed as follows.

1. The predicate `inv_itemmaping` is called (Figure 4.10 (1)).
2. The term `inv_itemmap(NAME)` is unified with the head of another rule (Figure 4.10 (2)).

[6] This is one of the reasons why we are able to first do the inverse mapping, and then perform the visualization again to achieve beautification; the beautified PR is in effect the 'exact' visualization of a given ASR data, whereas the edited picture just prior to beautification is not.

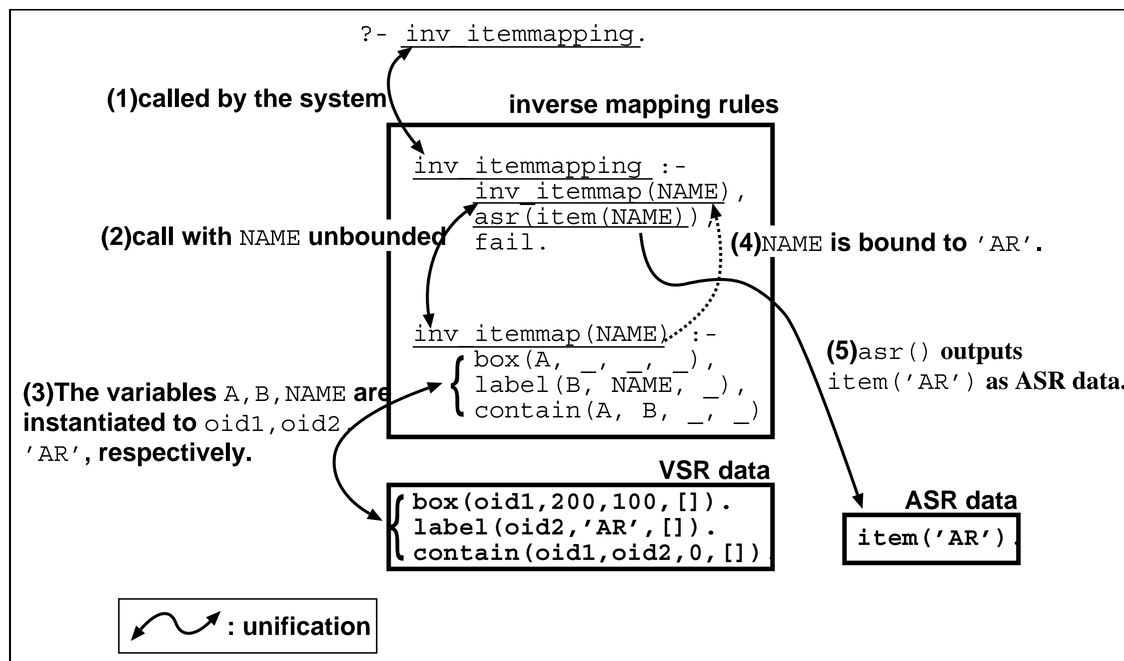


Figure 4.10: The execution of inverse mapping rules that translate a set of of VSR data (box, label, and contain) into ASR data (item).

3. The three terms (box, label, and contain) are unified with a set of VSR data in database, and the variables A, B, and NAME are instantiated (Figure 4.10 (3)).

4. The variable NAME is bound to 'AR' (Figure 4.10 (4)).

5. The predicate asr/1 generates item('AR') (Figure 4.10 (5)).

Since the predicate fail forces backtracking, all combinations of VSR data (box, label, and contain) are tested whether they match with the body of the rule, thus all ASR data are generated by the inverse visual mapping. Similarly, order(ILIST) is derived from the horizontallisting relation in the VSR database, and output in a certain format. Compared to the visual mapping rules (Figure 4.3), we can see that each clause corresponds one-to-one, and the terms in the clause body and their ordering are almost the same. Derivation of other inverse mapping rules is similar.

Often diagram interpretation involves a multiplicity of interpretations. In TRIP2, diagram is interpreted with inverse mapping rules. Thus the problem of multiple interpretations can be solved by writing the rules carefully. When the mapping rule allows multiple interpretations, the ASR data of all interpretations are usually generated, because most inverse mapping rules search with backtracking like the rules in Figure 4.9.

As mentioned earlier, the predicate inv_itemmap has essentially the same structure as itemmap, but the width and height of the box are ignored. If one desires to restrict the size of boxes within a certain range, only the predicates that specify the range of the width and height need to be added as follows:

```
inv_itemmap(NAME) :-
    box(A, W, H, _), W < 300, H < 200,
    label(B, NAME, _),
    contain(A, B, _,  _).
```

4.3 Examples

In this section, we show the examples of TRIP2 in action. All the figures in this section show the screen dumps of two windows of TRIP2. In the following examples, we focus on the feedback of user's operations on the PR data. Appendix C shows visual and inverse visual mapping rules for the examples in this section.

4.3.1 A Simple Graph Editor

This example is a simple graph editor on TRIP2. Appeindix C.1 shows the visual mapping rules for this example: there, nodes are mapped to circles, and edges are represented as straight lines. The user of this editor simply draws circles, lines, and labels. These pictures are recognized as a graph, and translated into `nodes` and `edges`. The nodes in this example are arranged by the `adjacent` relation which is an entry to the graph layout constraint solver in TRIP2. The graph drawing algorithm used here is described in [65, 66].

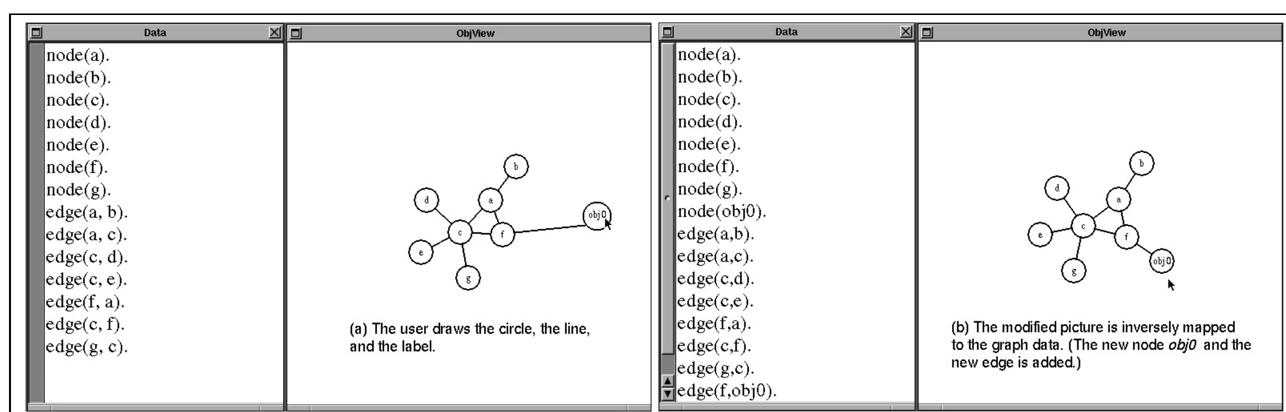


Figure 4.11: A simple graph editor.

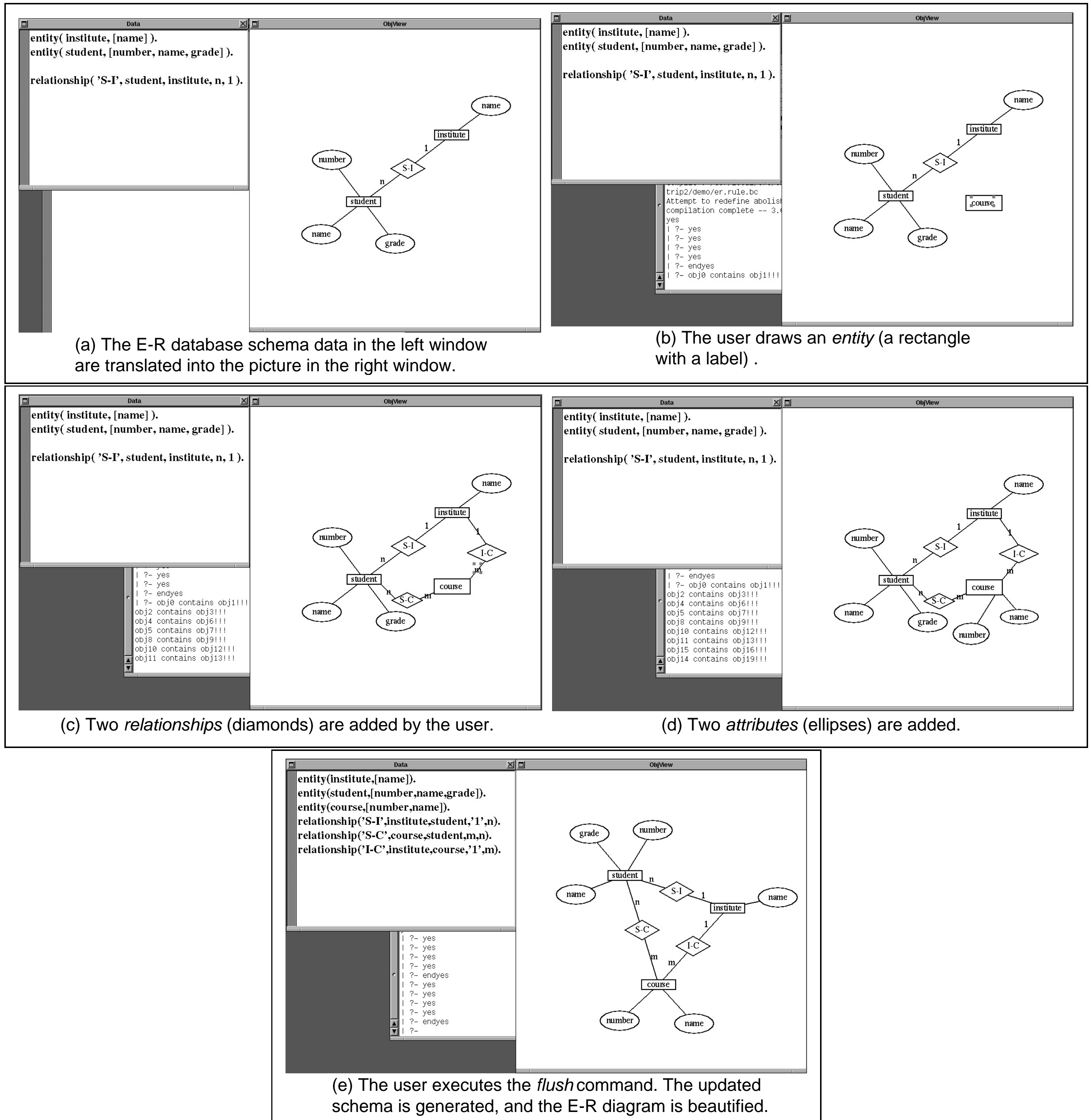
4.3.2 A Simple E-R Diagram Editor.

Figure 4.12 shows the screen snapshots of the windows of an Entity-Relationship Diagram editor. The upper-left window shows the E-R database schema in a textual form, and the right window displays its E-R diagram. The user extends the schema by adding a new entity *course* (Figure 4.12(b)) with two attributes (*number* and *name*) (Figure 4.12(d)), and new relationships (*S-C* and *I-C*) (Figure 4.12(c)). The user then performs the *flush* command. The modified picture in the right window is translated back to the E-R schema data in ASR, and these data are translated into the corresponding beautified E-R diagram (Figure 4.12(e)).

In this example, entities (represented by rectangles), and attributes (represented by ellipses) are arranged by the graph layout constraint solver as is with the previous graph editor example.

4.3.3 A Small Othello Game

This example provides an interface for a small Othello game application. Figure 4.13 shows the screen dumps of the windows during the game, which display the board data (ASR) and its pictorial representation. The Othello application itself is written in Prolog, and uses this ASR data directly as its representation of the board (therefore, there is no translation process between AR and ASR). It



(a) The E-R database schema data in the left window are translated into the picture in the right window.

(b) The user draws an *entity* (a rectangle with a label) .

(c) Two *relationships* (diamonds) are added by the user.

(d) Two *attributes* (ellipses) are added.

(e) The user executes the *flush* command. The updated schema is generated, and the E-R diagram is beautified.

Figure 4.12: A simple E-R diagram editor.

has no knowledge of the interface of the game. The board data consists of six rows of six characters (Figure 4.13(a)). The character '*e*' means that the position is empty. The characters '*b*' and '*w*' represent black and white stones respectively.

Player-1 first draws a circle with its color attribute black, which represents a black stone (Figure 4.13(b)), and executes the *flush* command. The modified picture is translated into the board data. Then, the application reverses the intermediate stones. The resultant ASR of the board is again visualized to update the PR (Figure 4.13(c)). Player-2 then draws a white stone (Figure 4.13(d)). This modification is also fed back to the board data (Figure 4.13(e)).

4.4 Related Work

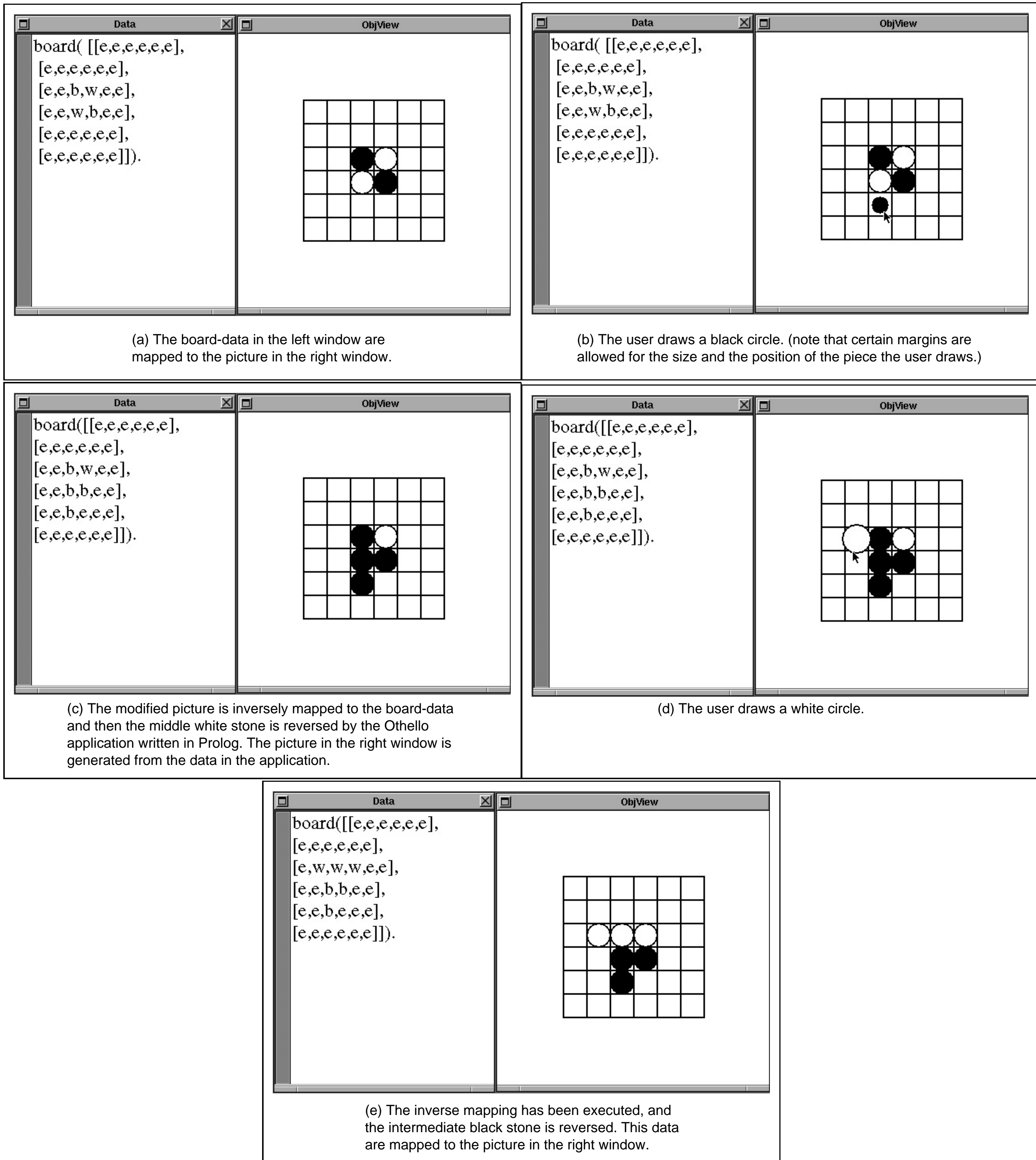
The work described in this chapter can be largely categorized into two research areas: one is data visualization, and the other is recognition of pictures and feedback of user's operations on pictures.

Visualization of *numerical* data has been studied intensively: Scientific visualization is now one of the most important areas of numerical data visualization — powerful personal tools such as *Mathematica*[129] are now in widespread use. Business graph packages are also used extensively in office and personal work. One interesting work is by Mackinlay[79], in which effective graphical presentations (such as bar charts, scatter plots) of relational information are automatically combined and designed. Although TRIP2 is not primarily intended for numerical visualization[7], a similar issue would arise as to which visual mapping rules should be selected and combined to provide an effective interface for complex application data. We are examining this topic as one of the current research issues.

There are a number of researches on visualization of non-numerical data as well. For example, visualization of programs [70, 93, 123], and visualization of database schema [130] has been studied. Here, the advantage of TRIP2 (and TRIP) is that it is a general-purpose system that can be applied to numerous types of visualization simply by defining declarative mapping rules. Declarative approach to program visualization is proposed by Cox and Roman[103, 28] in the system *Pavane*. Pavane provides three-dimensional animation of concurrent program written in the *Swarm* notation via the specification of declarative rules. The difference from TRIP2 is that (1) it is able to handle temporal relations for animation, but (2) it does not use constraint satisfaction techniques for layouts, and (3) it does not support inverse mapping from pictures to programs. In the context of TRIP, a general framework for visualization of abstract data and relations that can be mapped to ordering relations among the constituent objects is presented in [92].

There are several researches on establishing general schema for feedback of user's operations on pictures. In the area of graphical UI, visual and/or demonstrational specification of how the primitive interface objects should look, feel, and work is possible with systems such as Peridot[90], and Interface Builder[95]. *Visual programming* systems also recognize user's operations, but are special-purpose in the sense that the interfaces are usually tailored for a single (visual) language; they are surveyed in [21, 110]. Of special interest are GRAFLOG[99] which is an interactive graphics interface in which drawings receive linguistic interpretations, i.e., a certain kind of visual language, PAM[74] which is a system for manipulating text-graphic patterns, and Golin's Palette system[44], which is a tool for constructing visual program editors using the Indigo language, which is a visual language for controlling a visual program editor. Constraint-based graphical UI systems such as Juno[94], ThingLab[11], ThingLabII[80], Metamouse[32] allow the users to specify the constraints among the picture objects, but do not explicitly support the specifications of mappings between application and picture data. Advanced UIMSs such as Garnet[91] and Unidraw[124] provide the

[7] It is of course possible and useful to apply TRIP2 to visualization of numerical data.



(a) The board-data in the left window are mapped to the picture in the right window.

(b) The user draws a black circle. (note that certain margins are allowed for the size and the position of the piece the user draws.)

(c) The modified picture is inversely mapped to the board-data and then the middle white stone is reversed by the Othello application written in Prolog. The picture in the right window is generated from the data in the application.

(d) The user draws a white circle.

(e) The inverse mapping has been executed, and the intermediate black stone is reversed. This data are mapped to the picture in the right window.

Figure 4.13: TRIP2 othello.

support for visual interfaces which reflect application semantics, but the schema includes more procedure-oriented and/or library-based specifications.

The approach taken by TRIP2 is quite different; a set of declarative mapping rules are only necessary to provide direct manipulation interfaces for multitudes of application data. This is possible because TRIP2 is based on our proposed model of general bi-directional translation between application and picture data.

The objective of declarative specification is also achieved with CONSTRAINT[131] with a schema called *constraint grammar*. The major difference with TRIP2, however, is that the data in an application object are accessed directly, rather than to provide the layers of abstractions as is with TRIP2. Therefore, an application and its interface become strongly interdependent with CONSTRAINT, making interface specification and modification more complex compared to TRIP2.

4.5 Conclusions and Future Work

We have presented an overview of how direct manipulation can be achieved in our proposed bi-directional translation model, and have described its prototype implementation, TRIP2, and the examples of its use. In all the examples, by specifying only the declarative rules, the users can directly modify a picture, and the manipulations are fed back to abstract structure data automatically by the system. We stress that, since the system-provided user interface is independent of ASR data in TRIP2, (1) the system is applicable to a variety of types of ASR data, and (2) different visualization and its DM interfaces can be specified for a single type of ASR data, by merely providing the appropriate mapping rules. Thus, we have made substantial progress in achieving *dialogue independence* in the context of DM-style interfaces, a task which was pointed out as being significantly difficult in [48].

Our experiences with TRIP2 also made us aware of some new research issues. First is to improve the turnaround time of feedback. In the early implementation, it takes several seconds to update the display in response to the user's operation. For the example of kinship diagram, the turnaround time is about 3 seconds, and for the Othello game, the turnaround time is about 5 seconds. Most of the time is spent for the execution of Prolog and the overhead of interlingual transfer of the data. The later implementation is improved by the use of the faster Prolog system[8]. It is compiled with TRIP2 into one application so that the time taken for the communication between Prolog module and the Objective-C module become much shorter.

Another way of tackling this problem is to achieve *Incremental translation*. At present, the translation process is *total* at each stage (except for spatial parsing), meaning that the entire data in each representation is translated into the next representation. Since changes are usually limited to some local part of the application data or the picture, incremental translation at each stage would substantially decrease the amount of computation required. The last strategy is being tested with the DeltaTRIP system[105] implemented and running on ParcPlace Smalltalk v2.5 (Figure 4.14), although it does not support inverse translation from pictures to application data. The differences with TRIP2 are that (1) the Prolog interpreter is now an integral part of the system, being able to manipulate Smalltalk objects, and (2) the translation process at each stage is incremental as discussed above. (The underlying incremental constraint solver is DeltaBlue[41], courtesy of John Maloney.) Compared to TRIP2, preliminary tests have shown substantial increase in the speed of visualization when the ASR data to be visualized is modified incrementally (each incremental translation process takes less than a second). The speed improvement achieved in DeltaTRIP is

[8] We used SICStus prolog[111].

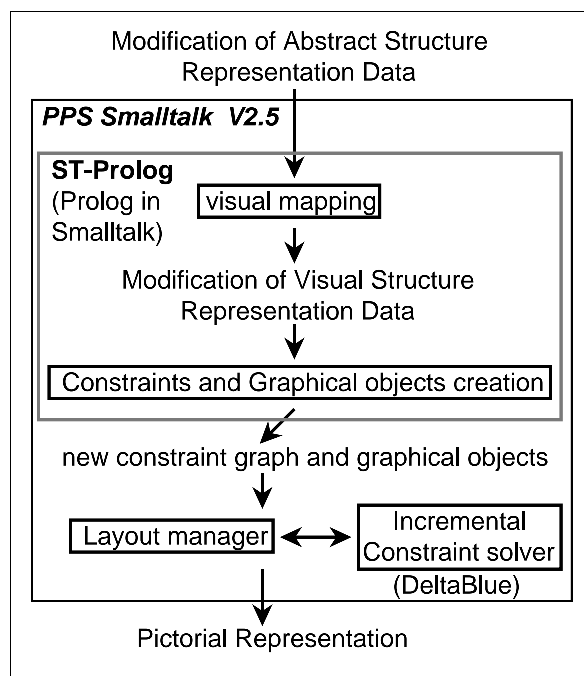


Figure 4.14: The DeltaTRIP system.

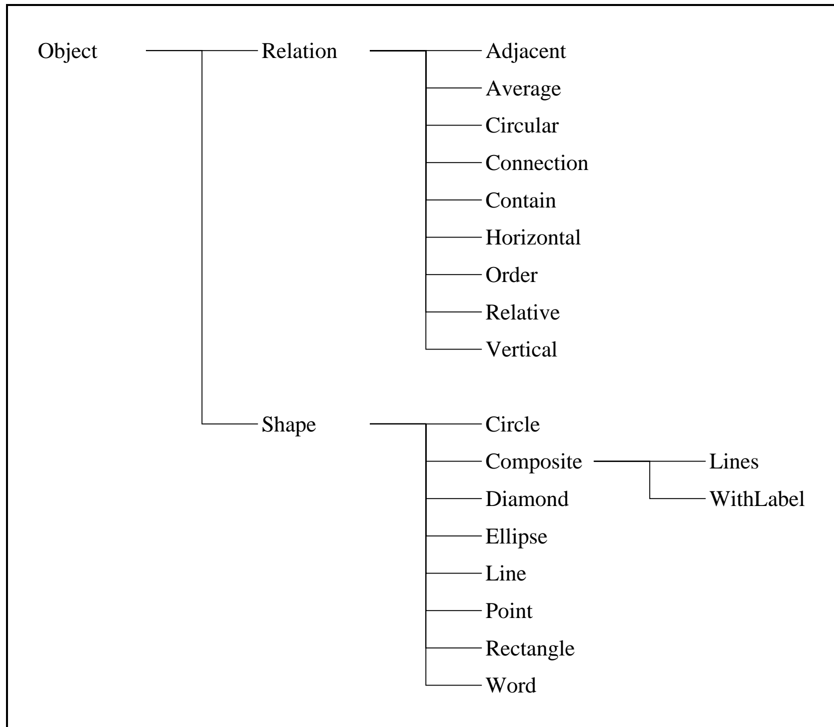
very promising, because the Prolog interpreter used in DeltaTRIP, being written in Smalltalk, is even an order of magnitudes slower compared to the one employed in TRIP2.

Second issue that we are facing is real-time and semantic feedback problem. Current implementation does not constraint any users' actions on the editor. However, it is necessary that the system can restrict some users' actions, or constraint the behavior of the part of a diagram. In addition, these restrictions may change by application's state. Because our bi-directional translation model separates user interface part and application part, such changeable restriction may be difficult to handle. However, constraining the behavior of a diagram by constraints in VSR can be possible. Furthermore, as the inverse mapping can interpret semantically illegal but syntactically legal actions, the application is able to detect such actions, and cancel it by not changing the application's data. There are many cases that might be enough.

Another way of extending TRIP2 is to use a constraint (logic) programming language, such as PROLOG III[25] and CLP(R)[59]. The numerical constraint solver in TRIP2 is external to the Prolog in the current implementation. As a result, it is difficult to write mapping rules which the high-level mapping rules are re-applied when constraints can not be solved in the lower translation stages. By integrating the constraint solving stages into logic programming with a constraint programming language, such mapping rules would be possible, allowing specification of more elaborate visualization. Such a language would also be beneficial for the inverse translation.

It is also important to integrate TRIP2 with existing UIMS or toolkits. Although TRIP2 has realized easy creation of direct manipulation interfaces, there are many functions that TRIP2 does not support. For example, TRIP2 does not intended to support ordinary widgets, such as menus, scroll-bars, and buttons. Business graphs and scientific visualization are not within the scope of TRIP2, too. TRIP2 must be integrated as one of the tools for creating user interfaces.

4.6 Classes in TRIP2



Classes for graphical relations are derived from the *Relation* class, and classes for picture objects are derived from the *Shape* class. Picture objects (in PR) are also used as graphical objects (in VSR) in the interaction module.

Chapter 5

TRIP2a — Constructing Algorithm Animations

5.1 Introduction

Animation is an effective technique for use in a graphical user interface (GUI), and is useful for indicating various changes, processes, and movement of data or execution of application programs. Animation is widely used in entertainment, with video games and cartoons making effective use of computer animation. The results of computations in various scientific fields, such as simulations of fluid mechanics, are often visualized and animated. Algorithm animation, which illustrates how a target algorithm works, is a useful tool for teaching algorithms, as well as for creating and debugging algorithms.

Nevertheless, these advantages are not fully utilized in current GUIs, because of the time and cost associated with creating animations despite the many tools available. The primary reason for the time and cost is that the process of creating animations is still procedural, and the programmer has to be concerned with various details of object movements, etc. As a result, although computer and graphics performance have improved dramatically, we cannot fully utilize this performance for visualization and animation in GUIs.

This chapter describes a tool for creating animations in a fully declarative framework to reduce the cost of creating animations and to facilitate their use. However, we do not attempt to cover all conceivable kinds of animations here. Our target is *abstract* animations; that is, we do not intend to deal with animations that illustrate "real" shapes of some real objects, nor the movements based on certain physical phenomena. For example, animations that represent the results of simulations of hydromechanics, animations of the human body, or animations of dinosaurs are outside the scope of this thesis. To be more precise, we will discuss animations that describe the process of changing abstract data or relations, such as algorithm animations or program visualizations.

Our target animations have two features in common:

1. The abstract data and the relations to be animated do not have their own intrinsic "shapes." How to visualize and animate them should be specified in some way so that they can be understood easily. This is in marked contrast to animations of real objects, which have their own shapes, and usually images of such objects should resemble their real shapes.

2. The abstract data are stored within the application program, and change during its execution. To create an animation of an application program, we have to obtain up-to-date information regarding the status of the application program during its execution.

TRIP2 described in Chapter 4 enables programmers to easily create direct manipulation-style interfaces for manipulating abstract data. In this model, visualization of application data and recognition of pictures are regarded as translations from/to application data to/from pictorial data via abstract structure representation (ASR) and visual structure representation (VSR). This translation is specified by a declarative mapping rule between ASR and VSR (*Visual Mapping Rule*). We have extended this bi-directional translation model to handle animations. The extended model includes the notion of time; i.e., it incorporates a series of changing data. Animations are achieved by interpolating a series of pictures *translated* from a series of application data. In addition, we define *operations* on four data representations in our model. These operations change their data to the next state. The picture changes, animations, are also regarded as operations on pictures, and how an animation works can be altered by providing a *transition mapping rule* that maps *abstract operation* on ASR to *transitional operation* on VSR. Using this model, only two sets of mapping rules, i.e., visual mapping rules and transitional mapping rules, are necessary to create an animation. Complex animation techniques, such as slow-in-slow-out and squash-and-stretch[76], can be specified easily in our model.

This chapter describes a prototype system based on this extended model. This system uses a visual mapping module similar to TRIP[65, 67], TRIP2[85, 120], and TRIP3[88, 87]. Thus, the same visual mapping rules can be used in our animation system. In particular, using the TRIP3 system, users can easily create visual mapping rules by example for this animation system.

The rest of this chapter is organized as follows. Section5.2 explains how the programmer makes an animation, and Section5.3 describes the implementation details of our system. Section5.4 shows various examples of use of our prototype system, and conclusions are presented in Section5.6.

5.2 How to Construct an Animation — Insertion Sort Example

To create an algorithm animation with TRIP2a, programmers have to write the following:

Visual mapping rules Visual mapping rules are used for translating ASR to VSR. They specify how to visualize application data; i.e., they specify how to lay out graphical objects in the target animation (Figure 3.5(a)).

Transition mapping rules Transition mapping rules are used for mapping abstract operations to transitional operations. They specify how to move graphical objects in an animation (Figure 3.5(b)).

Annotations Annotations output abstract operations when an application executes its corresponding operations. They are used for extracting information from the executed application program, and can be regarded as specifications of translations from applications' operations to abstract operations (Figure 3.5(c)).

For example, to create an animation of an insertion sort algorithm, a programmer must write an insertion sort program, and annotate it so that the information on the execution of that program is obtained. Figure 5.1 shows an insertion sort program written in Prolog. Annotations are inserted at points where the application executes an interesting algorithm operation. In this example, after performing `insert/3`, `write_asr/2` outputs ASR data such as:

```
numlist([3,8,4,9],[1,2,5,6,7]).
```

The first argument of `numlist` represents numbers to be inserted, and the second argument represents already sorted (= inserted) numbers. At the same time, the predicate `write_ao/1` outputs an

```
% For demo
go :- tell('sort.anim'),
      % Write Initial State
      write_asr([5,2,6,7,1,9,4,8,3],[]),
      write('%'),nl,
      % Start Sorting
      sort([3,8,4,9,1,7,6,2,5], L),
      told.
% Sorting Program – Insertion Sort
sort(I, O) :- sort1([], I, O).
sort1(_, [], []).
sort1(L, [IH|IT], O) :-
      % Sort IT
      sort1([IH|L], IT, TMP),
      % and Insert a number to the sorted numbers
      insert(IH, TMP, O),
      % Output ASR data and Abstract Operations
      write_asr(L, O), write_ao(IH).
% Insert N to the sorted list
insert(N, [], [N]).
insert(N, [F|S], [N, F|S]) :- N =< F.
insert(N, [F|S], [F|L]) :- insert(N, S, L).
% Program for output ASR and AO
write_asr(L1, L2) :-
      reverse(L1, L3),
      write(numlist(L3,L2)), write('.'), nl.
write_ao(O) :-
      write(insert(O)), write('.'), nl,
      write('%'), nl.
```

Figure 5.1: Annotated insertion sort program.

abstract operation `insert(N)`, which means that the program inserts the number `N`. The character `%` output just after `insert/1` separates these data from those generated at the next execution of `insert/1`. This output is logged to a file (Figure 5.2), and used to create an animation of this execution.

Then, the programmer writes the visual mapping rule shown in Figure 5.3 to visualize the ASR data `numlist()`. There are two parts to this rule. In the first part, the predicate '`o`' translates all the numbers in `numlist/2` into `box/4` using `num2barmap/1`, and in the second part, the predicate '`r`' translates `numlist/2` into three `horizontallisting/3`, which arranges boxes horizontally with appropriate intervals.

Last, the transition mapping rule shown in Figure 5.4, which specifies that the inserted `object(X)` should move clockwise, is provided.

```
numlist([3,8,4,9,1,7,6,2,5], []).
%
insert(5).
numlist([3,8,4,9,1,7,6,2], [5]).
%
insert(2).
numlist([3,8,4,9,1,7,6], [2,5]).
%
insert(6).
numlist([3,8,4,9,1,7], [2,5,6]).
%
 ..... omitted .....
%
insert(3).
numlist([],[1,2,3,4,5,6,7,8,9]).
%
```

Figure 5.2: Log file of the insertion sort program.

```
% Mapping Numbers to Graphical Objects
objectmap([o]).
o :- numlist(L1, L2),   % Get numlist/2
     num2barmap(L1),    % Map numbers to boxes
     num2barmap(L2).    % Map numbers to boxes
num2barmap([]).
num2barmap([B|L]) :-
     Y is B * 20,
     % a box with width=30 and height=Y
     box(B, 30, Y, [shaded]),
     num2barmap(L).
relationmap([r]).
r :- numlist([H|L1], L2),   % Get numlist/2
     do_layout([H|L1], L2),% Layout numbers
     place(H,80,100,[]).
% Layout numbers horizontally
do_layout(L1, []) :-
     horizontallisting(L1, 12, [bottom_align]).
do_layout([], L2) :-
     horizontallisting(L2, 12, [bottom_align]).
do_layout(L1, L2) :-
     horizontallisting(L1, 12, [bottom_align]),
     horizontallisting(L2, 12, [bottom_align]),
     tail(L1, T), L2 = [H|_],
     horizontallisting([T, H], 50, [bottom_align]).
```

Figure 5.3: A visual mapping rule for insertion sort animation.

```
transitionmap([t]).
% insert(X)  =>  move(X,[clockwise])
t :- insert(X),              % get insert()
     move(X, [clockwise]).   % output move()
```

Figure 5.4: A transition mapping rule for insertion sort animation.

Figure 5.5 shows the resulting animation created with the above mapping rules[1]. Note that not only the inserted bar is moving; the other two bars are also being moved to make room for the inserted bar. The programmer did not describe such movements, but the system automatically created the transitions for the movements of the two bars (See Section 5.3.1).

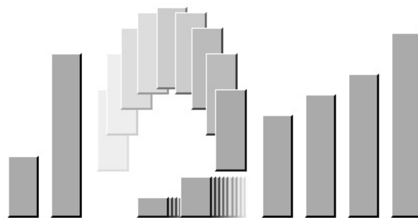


Figure 5.5: Animation of an insertion sort algorithm.

5.3 Implementation of TRIP2a

We have implemented a prototype system based on the extended bi-directional translation model. This system is integrated with TRIP2 [85, 120] and shares the TRIP module and the interaction module. This system was written in Objective-C, and was created on NextStep[2] using the NextStep

[1] Our system uses color, but most screenshots in this thesis have been converted to gray-scale pictures. In addition, some figures show trails of objects, which are displayed only to illustrate the animation and are not actually presented in the real system.

[2] NextStep is the operating system developed by NeXT computer.

Application Kit[3] and Interface Builders. This section describes how we generate pictures intermediate between those generated from the application data, what kinds of transitional operations (including various animation effects) have been built, and how we obtain data from a running application program.

5.3.1 Implementation of Animations

In our implementation, a picture is represented as a set of picture objects in Objective-C. Thus, an animation is a set of transitions from the objects in one picture to those in another picture. An animation from picture A to picture B is created in the following manner:

1. Find correspondences between the objects in picture A and the objects in picture B, and examine how the object in picture A is changed to the corresponding object in picture B (Figure 5.6(1)).

2. Make instance objects of Transition class with the above pairs of corresponding objects, which will change the object in A to that in B (Figure 5.6(2)).

If the corresponding object is not found, the system will make a transition from/to the object to/from a *null* object (an object whose size is zero).

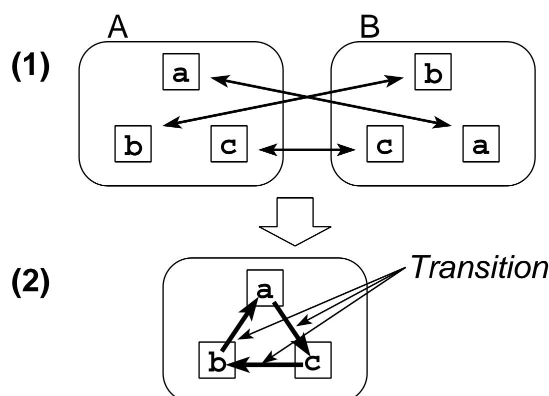


Figure 5.6: Making an animation.

Correspondences between objects in two pictures are searched for using the names of the objects. For example, if `objX` in picture A and `objY` in picture B have the same name, it is assumed that they are the same object, and that `objX` is moved and/or resized to `objY`. This means that to animate an object, the programmer must maintain its name in a sequence of pictures; i.e., the programmer has to name objects in a consistent way. The names of objects are specified in visual mapping rules.

Other animation systems often require explicit specification of transformations of graphical objects in an animation. For example, to generate animations using Pavane[103, 28], the programmer has to write a rule that maps data in the old and current states of a program into graphical objects, which change over time. In our system, the transformations of graphical objects are detected automatically, and do not need to be specified. Programmers have only to write a rule to layout abstract data, and *alter* the method of transformation if the default method is not sufficient.

[3] NextStep application kit is the Objective-C class library for developing various software on NextStep.

One interesting feature of our method of creating animations is that the structure of the moving sub- picture is often preserved automatically. For example, in Figure 5.7, which shows an animation of a list structure, the layout of the sub-structure `((b) c d)` is maintained during its movement[4]. This is because the transitions of these cells and arrows are created using the same (straight) transitional operation, and the starting and the ending layouts of the sub-tree are the same. This kind of movement would be cumbersome to specify if the programmer had to specify the movements of every graphical object in the animation.

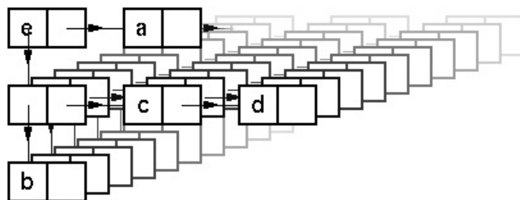


Figure 5.7: A cons-cell animation.

5.3.2 Specifying Transitional Operations

In our system, the motions of graphical objects are specified with transition mapping rules that translate abstract operations on ASR into transitional operations on VSR. These transitional operations are attached to each graphical object in the picture translated from ASR, and are used to make object transitions.

There are several ways to specify transitional operations:

1. *No specification*. When the programmer does not specify transition mapping rules, transitional operations for the graphical objects are determined automatically. For example, if a movement of a graphical object is not specified by the programmer, the transitional operations that move the object in a straight line and that scale linearly are used by default. Therefore, at the beginning of the construction of an animation, the programmer does not necessarily have to provide a transition mapping rule. The programmer can execute an animation without transition mapping rules, and may later improve it by specifying transitional operations.

2. *Using predefined transitional operations*. The following operations are provided:

   move Specifies how to move a graphical object. The following operations are provided:

   straight A straight path is created. The graphical object to which straight is attached moves in a straight line from the starting location to the ending location. This operation is used by default when no transitional operation is assigned to a moving object.

   clockwise, counterclockwise A path that goes clockwise/counterclockwise to the destination is created.

   lazy, immediate The lazy operation keeps an object from moving until the end of this transition. The immediate operation moves an object immediately at the beginning of this transition, to the final position and size.

   up, down, left, right These operations specify tangent vectors at the start and the end of a movement of a graphical object.

[4] Here, the list `(e a (b) c d)` is changed to `(((b) c d) a)`.

rotate Specifies how to rotate an object during movement. The direction and the number of rotations can be changed.

scale Specifies how to scale an object. immediate, lazy, etc., are provided.

color Specifies how to change the color of an object.

Furthermore, several decorative animation effects can be specified in the same way. The techniques of cartoon animation[76] are also important in graphical user interfaces[24]. The following are examples of these techniques:

blink The object blinks during the movement, or blinks before and after its movement.

shake The object trembles before moving so that the start of its movement can be easily recognized.

slow-in-and-slow-out This operation slows down a graphical object near the start and the end of its movement, which makes the movement seem more natural.

squash-and-stretch This operation squashes or stretches a graphical object as it moves. This effect can depict rigidity of an object, and makes the object more vivid.

Figure 5.8 is a screenshot of an animation of the tower of Hanoi. It shows the trajectory of the disk moved from the left tower to the right tower. This is an example of use of slow-in-and-slow-out and squash-and-stretch. The moving disk in the figure is specified by the following simple transitional operations:

```
move(X, [up,squash-and-stretch]).
move(X, [slow-in-and-slow-out]).
```

The disk moves upward at the start. The movement of the disk is slowed down near the start and the end, and the disk is stretched as it moves[5].

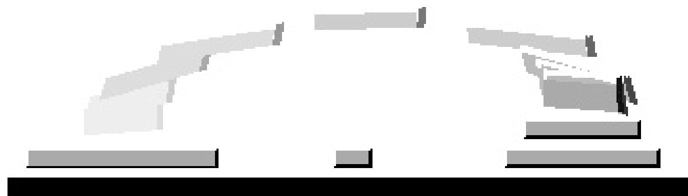


Figure 5.8: Slow-In-Slow-Out and Squash-and-Stretch.

3. *Defining transitional operations*. It may be convenient if the programmer can define transitional operations by writing functions that describe the motion of a graphical object. We are planning to implement this feature by using a fast constraint solver to evaluate these functions.

5.3.3 Application Interface

As mentioned above, the layout and the movement of graphical objects can be determined by specifying only two mapping rules; i.e., a visual mapping rule and a transitional mapping rule. However, to obtain information from the running application, some programs must be inserted into the target application program. At present, there are three ways to do this:

[5] Here, the object is stretched in proportion to its acceleration.

- Writing the operations and the data in the application to a log file as ASRs and AOs. This can be used for almost all programming languages as it uses only file operations.

- Sending abstract operations and data using the ToTripSpeaker class, which is a subclass of the Speaker class in the NextStep Application Kit. In this case, the execution of the application and its animation run concurrently. However, the application program must be written in Objective-C.

- Inserting the predicate intrEvent into the application program on ASR[6].

All of the above methods require modification of the application program, which is obviously undesirable. We are planning to use the technique employed in debuggers to obtain operations and data in the application without modification of the application program.

5.4 Examples

This section describes several examples using our system. Our system uses color, but all screenshots of the system in this chapter have been converted to gray-scale pictures. In addition, some figures in this chapter show tracks of objects, which are displayed to illustrate an animation but do not actually appear on screen. The mapping rules for all the examples in this chapter are listed in Appendix D.

5.4.1 Animations of Data Structures

Animating Graph Structures

As mentioned above, we have a special constraint solver for the layout of undirected graphs. Using this constraint solver, a graph structure can be visualized easily by providing only its nodes and edges[7].

In the animation shown in Figure 5.9, a new edge between `b` and `d` is added to the graph in Figure 5.9(a). In this transition, due to an error with our graph constraint solver, the positions of nodes `e` and `g` are reversed in Figure 5.9(c). However, using an animation, users would easily be able to see that the positions of the nodes are exchanged[8].

Animating List Structures

List structures are often illustrated as box-and-pointer diagrams. Using almost the same visual mapping rules listed in [65, 67], we have animated list structures.

Figure 5.10 shows an animation depicting the allocation of a new cell. The boxes that represent the new cell (`c`) and the arrow pointing to the cell are newly generated in this transition. They gradually become larger, increasing eventually to the size of the cell. We can also easily move the cells in the list together (Figure 5.11). In this animation, the list `(e a (b) c d)` is changed to `(((b) c d) a)`. The cells corresponding to `((b) c d)` are moved together to the destination.

[6]ASR is implemented as clauses in Prolog. To use this method, the program has to be written in Prolog.
[7]It is also possible to specify relative lengths of edges.
[8]The exchange of nodes is caused by the differences in the initial conditions of the graph constraint solver. Of course, these sudden changes of an image are undesirable. We are currently working to improve the algorithm of the graph constraint solver.

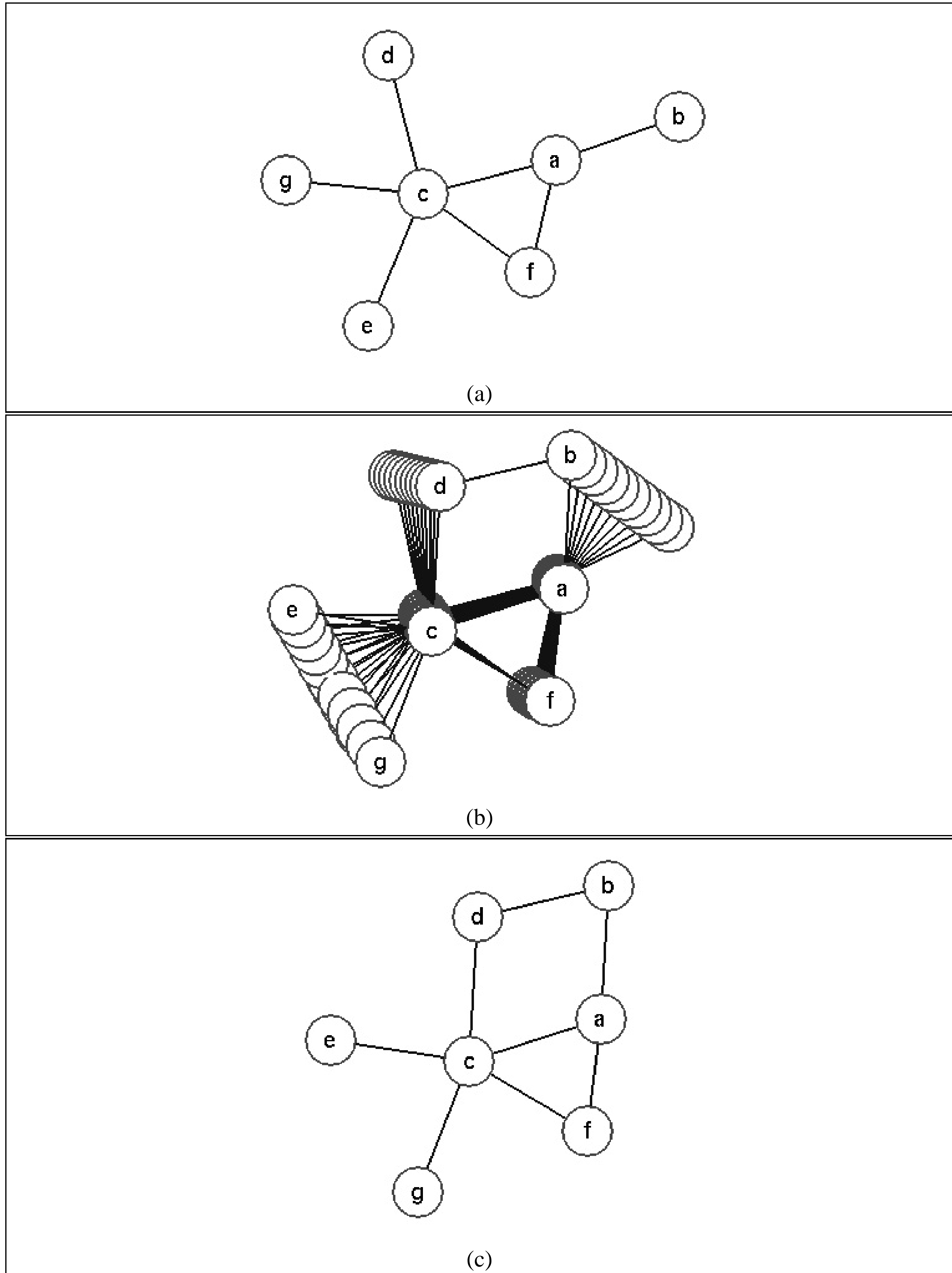


Figure 5.9: Animation of a graph structure.

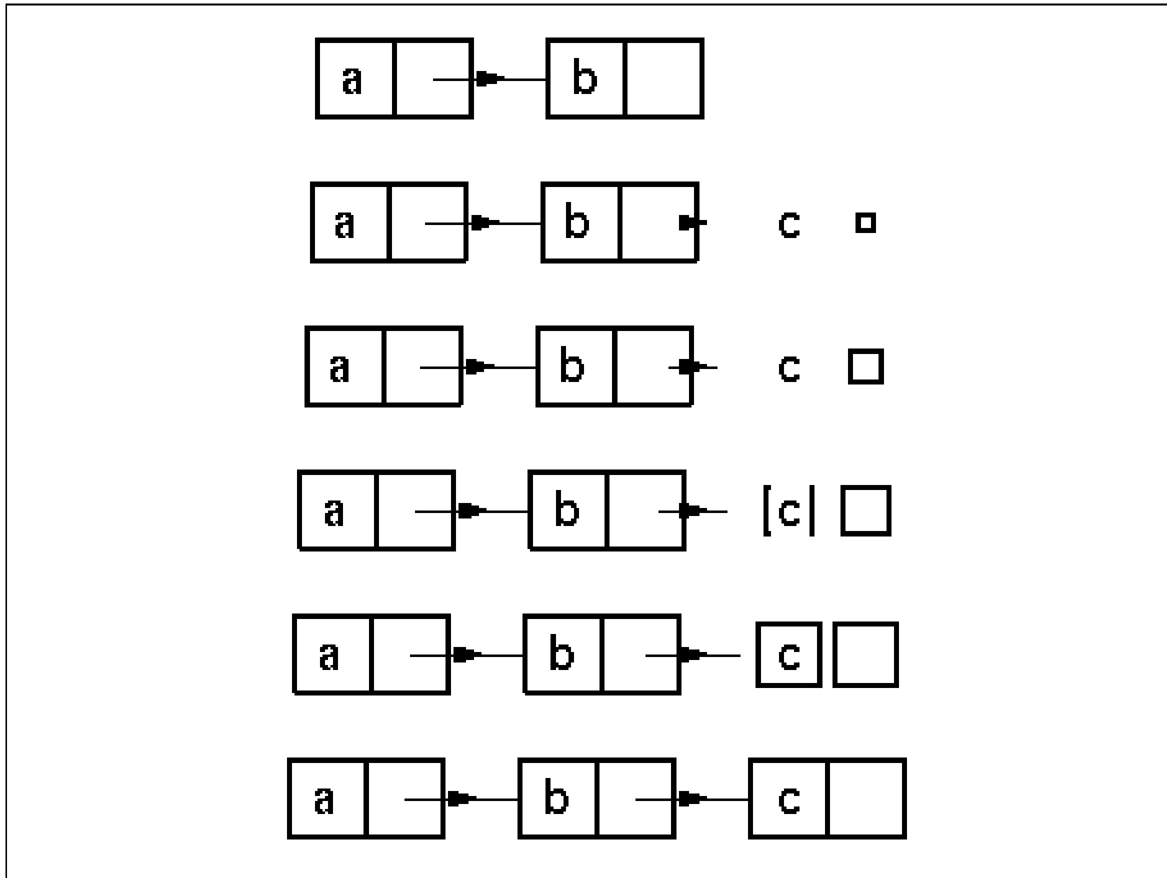


Figure 5.10: Allocation of a new cell.

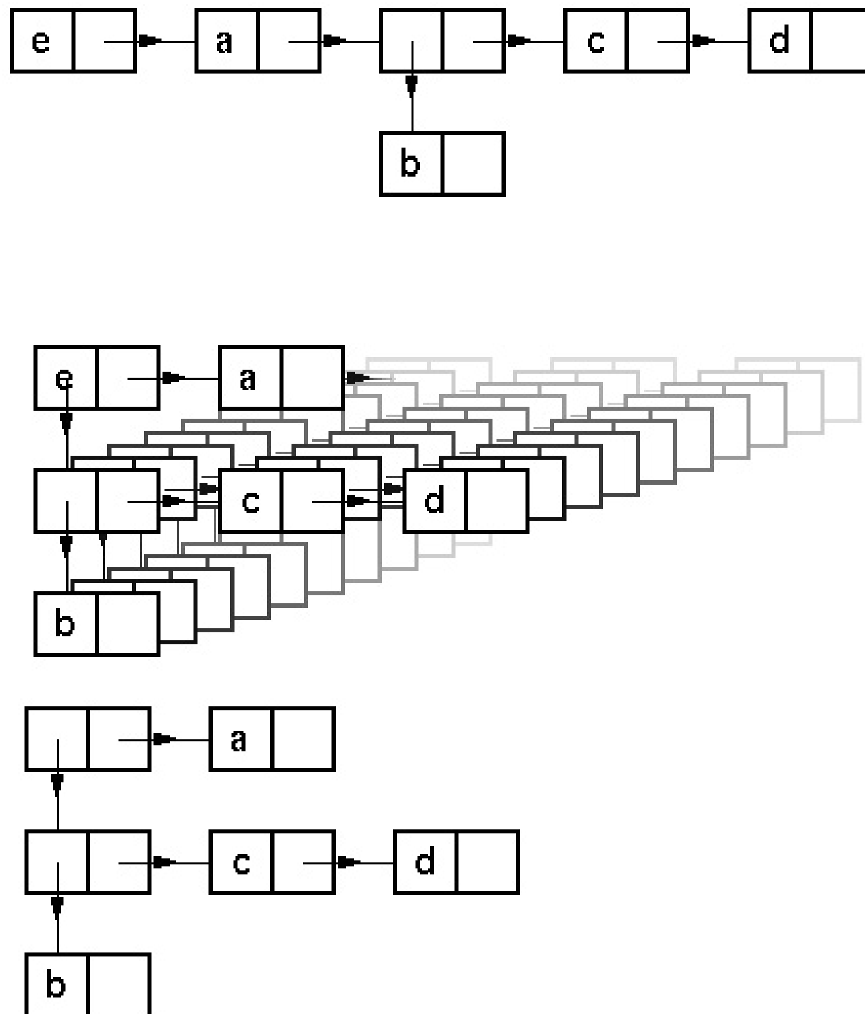


Figure 5.11: Moving the substructure of a list

5.4.2 Sorting Algorithms

Sorting algorithms are often used for algorithm animation systems. The previous section already described the insertion sort animation. This section describes a bubble sort animation, a quicksort animation, a merge sort animation, and a heap sort animation.

Bubble Sort

The bubble sort algorithm is a naive sorting algorithm, which searches for a maximum number in order. This algorithm is thus named because the process of searching looks like a rising bubble. Figure 5.12 shows animations that represent a “rising” number. To move bars linearly, the default transitional operation `straight` is used in this animation. Therefore, this animation is created using only a simple visual mapping rule that lays out bars horizontally.

Quicksort

Quicksort is based on the divide-and-conquer paradigm. The numbers to be sorted are put in an array, and divided into two parts so that the numbers in one part are smaller than a *key* and those in the other part are larger than the key. Then, the two parts are sorted recursively.

The dividing process is performed using two indices. In our quicksort animation, they are represented as two small black rectangles (Figure 5.13(a)). The bars between the two indices are to be partitioned. The number at the right end of the sorting region is used as a key, which is drawn in black.

The process of partition proceeds as follows. First, the left index searches for a number larger than the key to the right, which is represented in the animation as its movement to the right, and then the right index searches for a number smaller than the key to the left, which is represented in the animation as its movement to the left. When these two numbers are found (Figure 5.13(b)), they are exchanged (Figure 5.13(c)). Here, the `clockwise` transitional operations are attached to the two moving bars. This process is repeated until the indices meet each other. In this partition, as only one number is larger than the key, no more exchange is necessary. At the end, the key is exchanged with the number at the boundary of the partition (Figure 5.13(d)). Then, the smaller and the larger part are each sorted in order. Figure 5.13(e) shows a screenshot at the beginning of the sorting of the smaller part.

Merge Sort

Merge sort is also based on the divide-and-conquer paradigm. First, the numbers are divided into halves. They are sorted severally, and then *merged*.

The numbers in the two parts before merging are drawn as white rectangles bounded with black or gray (Figure 5.14 (a)). Merging is achieved by taking the larger of the two numbers at the right end of the two parts, and arranging it in a different place. In Figure 5.14(a), the number at the right end of the left part is larger, so it is moved above (Figure 5.14(b)). Next, as the number of the left part is larger, it is moved above and arranged at the left of the previous bar (Figure 5.14(c)). The bars that are moved above are represented as gray shaded rectangles. When merging is finished, the merged numbers are moved down (Figure 5.14(d)), and the next merging process is started. Figures 5.14(e) and (f) show the final merging process.

In this example, no transitional operation is specified. Thus, the transitional operation `straight` is automatically used, and the rectangles move linearly.

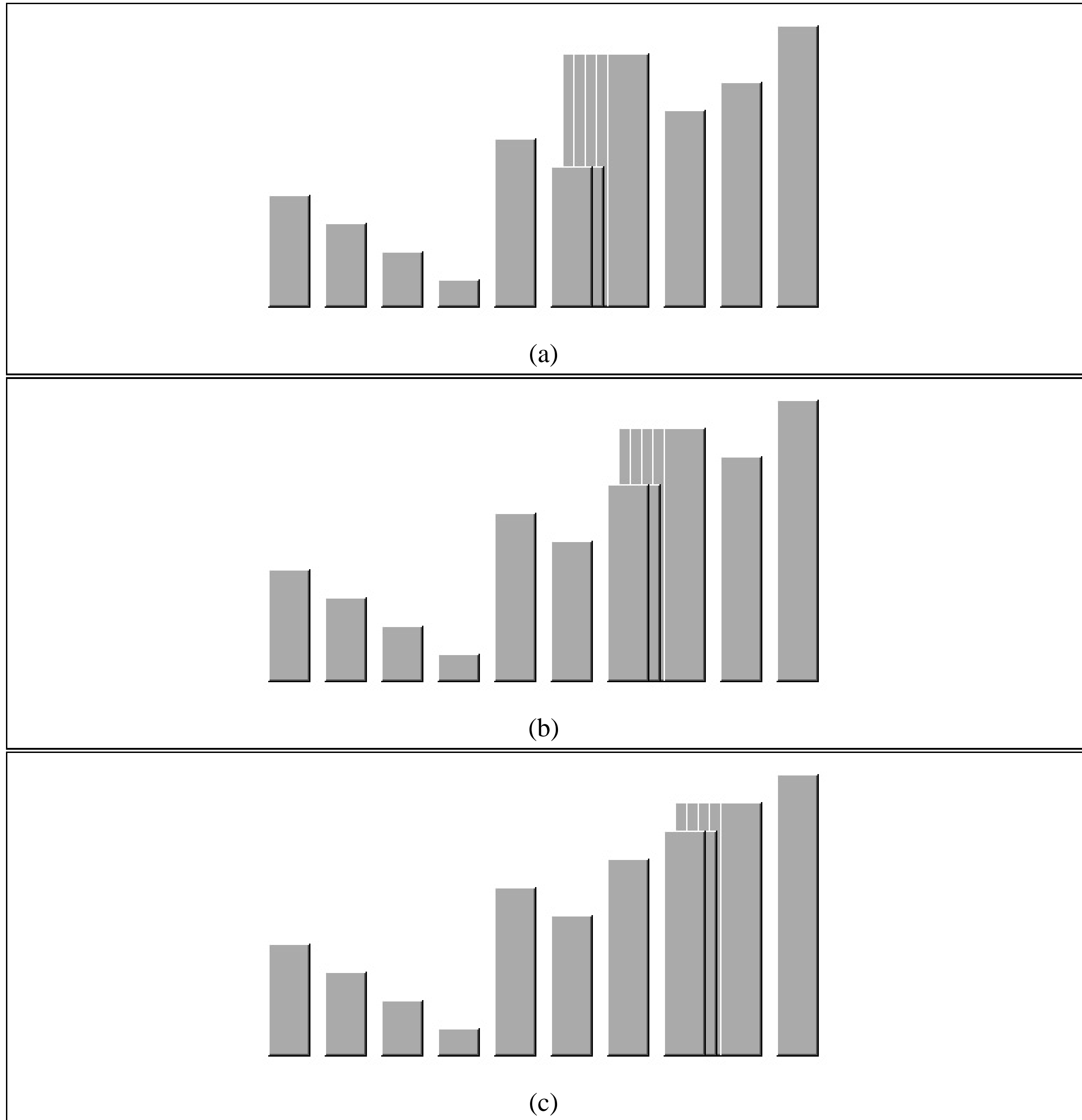


Figure 5.12: Bubble sort animation.

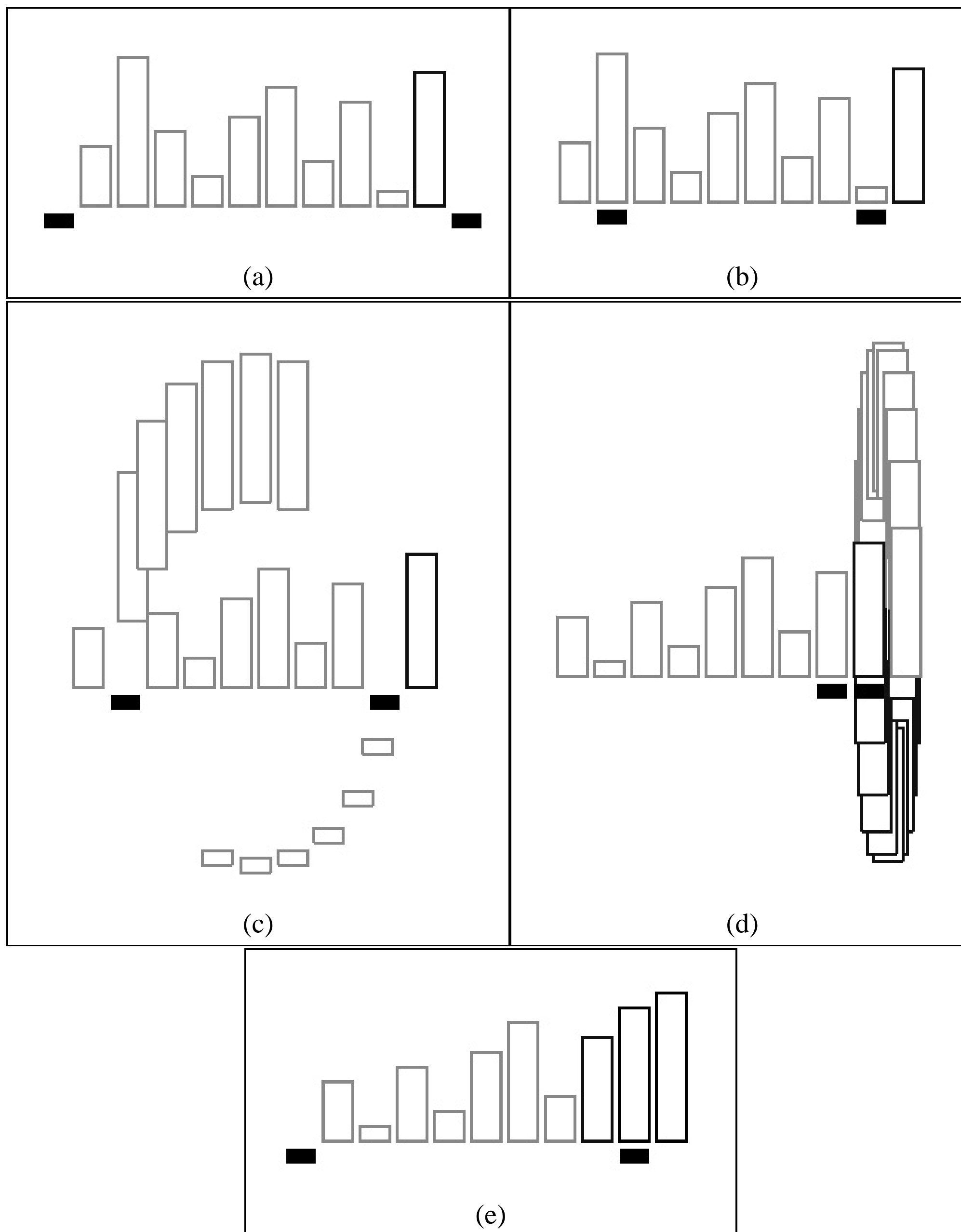


Figure 5.13: Quicksort animation.

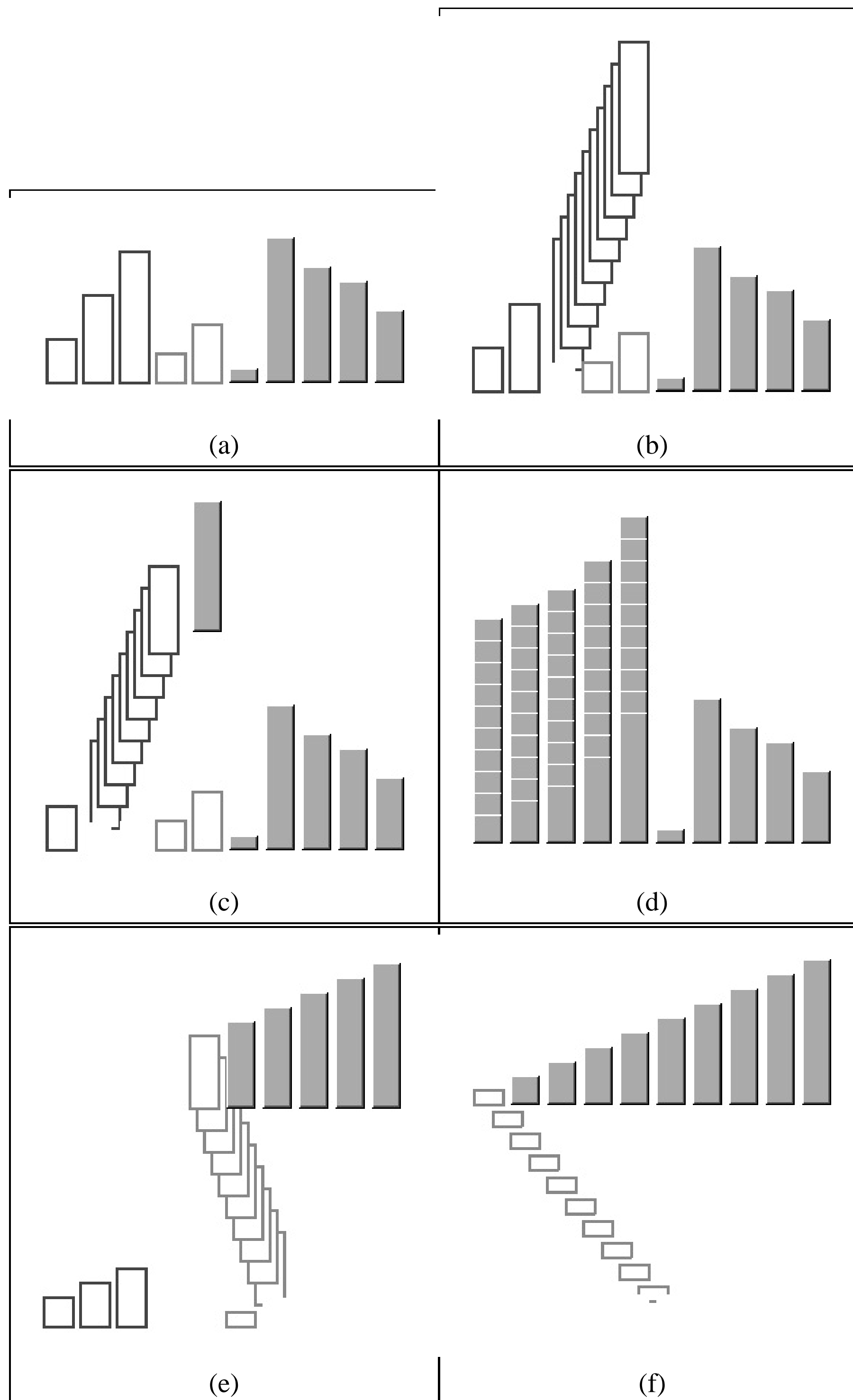


Figure 5.14: Merge sort animation.

Heapsort

In the heap sort algorithm, the numbers are put into a *heap*, i.e., a binary tree in which each node is larger than its two children. Therefore, the root node is the largest number in a heap. The heap sort algorithm proceeds by repeating the following procedure:

1. Creating a heap.
2. Exchanging the root node with the node at the end of the heap, and excluding it from the heap. That is, the largest numbers are removed one by one from the heap.

We visualized these heap structures as binary trees, and animated this algorithm.

For example, Figure 5.15 (a) shows a heap in the shape of a binary tree. First, the root node is exchanged (Figure 5.15 (b)). The exchanged root node is drawn in black. Then, the heap is restructured by moving the new root node down to the appropriate position (Figures 5.15 (c) and (d)). Again, the root node is exchanged (Figure 5.15 (e)), and the heap is restructured. Figures 5.15 (f) and (g) illustrate the final steps of this sorting procedure. To help identify the movements associated with removal of the max number and restructuring the heap structure, we used the clockwise movement to remove the max number.

The layout of a heap is specified by a visual mapping rule for ordinary tree structure data. The transitional operations are not specified in this animation. Only a visual mapping rule is provided to create this animation. In the same way, a number of animations can be constructed by modifying mapping rules for ordinary data structures, such as graph and tree structures. A number of mapping rules for various data structures are already available.

5.4.3 The Tower of Hanoi

The tower of Hanoi is a game played with three poles and a set of disks, in which the player tries to move the tower to another position by moving only one disk at a time; i.e., to move the tower shown in Figure 5.16 (a) to the position shown in Figure 5.16 (f). However, the player is prohibited from putting a larger disk on a smaller one as shown in Figure 5.16 (b).

We have animated the process of playing this game. Figures 5.16 (c) and (d) show the first two movements of a disk, and Figures 5.16 (e) and (f) show the last two movements. In this animation, the tangent vectors at the start and the end of a transition are specified as `up` and `down`, respectively, so that a disk first moves up and finally moves down at the destination. Note that we do not need to specify whether it should move left or right, and the destination is automatically determined by the next picture translated from the next state data.

5.4.4 Bin-Packing Problem

The bin-packing problem concerns finding the optimum way of packing objects into the minimum number of unit-size bins. In our animation, an object is drawn as a gray shaded rectangle (Figure 5.17). Bins are not visualized explicitly, but the height of the large background rectangle represents the size of the bins.

We implemented the *first-fit* approach, which means that an object is packed into the first bin that can contain it. For example, an attempt was made to pack the object placed at the left in Figure 5.17 (a) into the first bin (Figure 5.17 (b)), but this failed. This process was repeated at the next bin, and was successful (Figure 5.17 (c)).

As in the animation of the tower of Hanoi, the transitional operations `up` and `down` are specified for the tangent vectors of the transition of this animation.

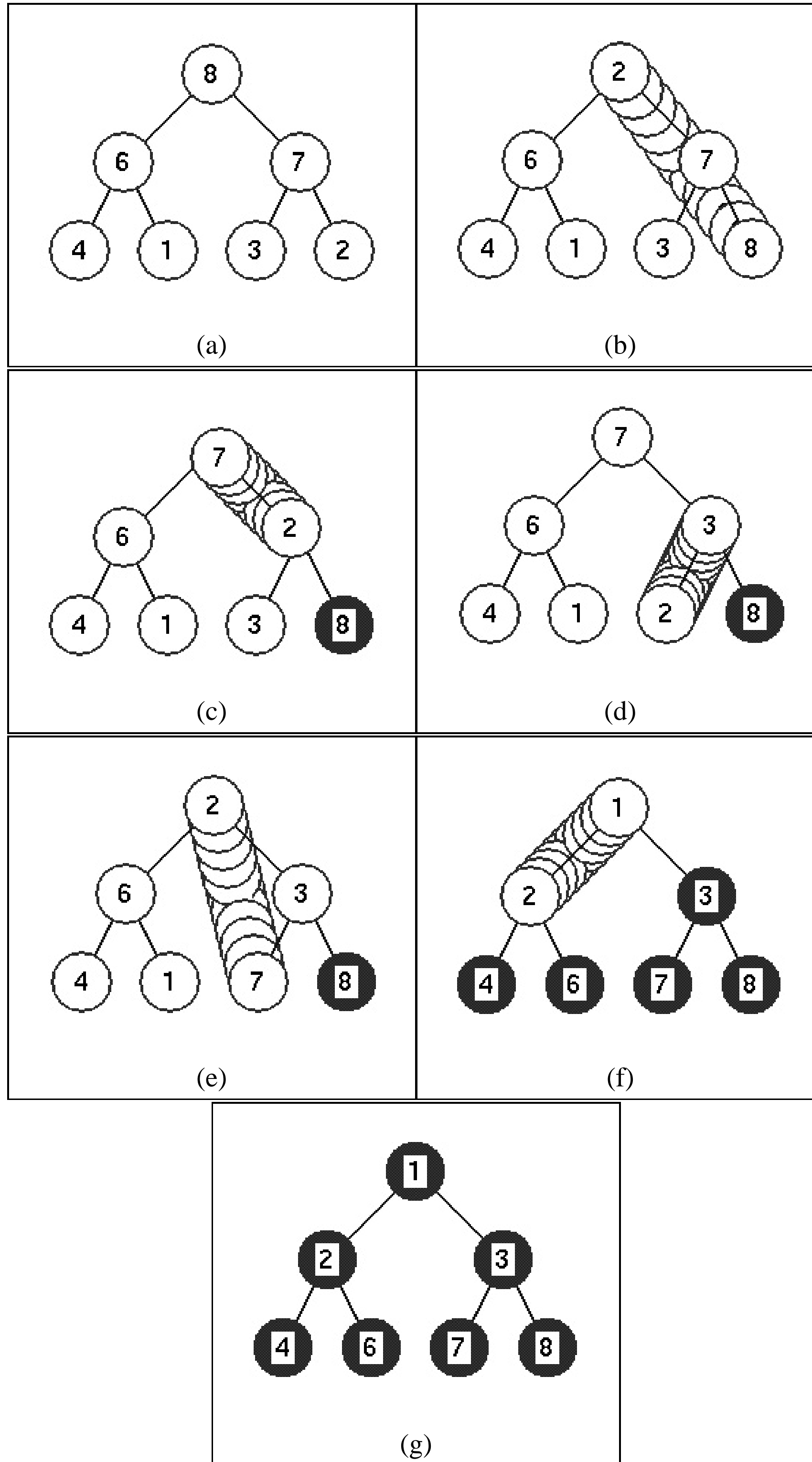


Figure 5.15: Heap sort animation.

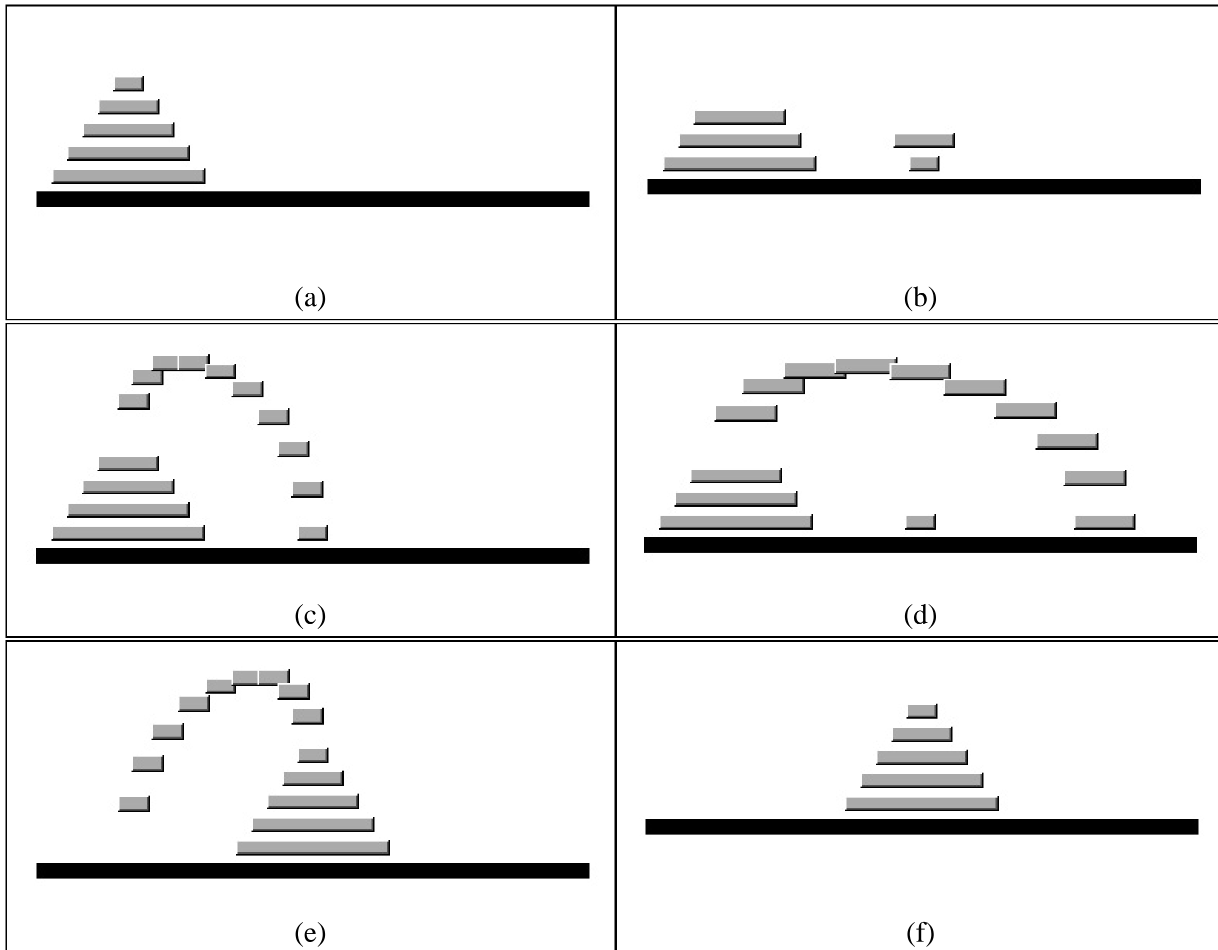


Figure 5.16: The tower of Hanoi.

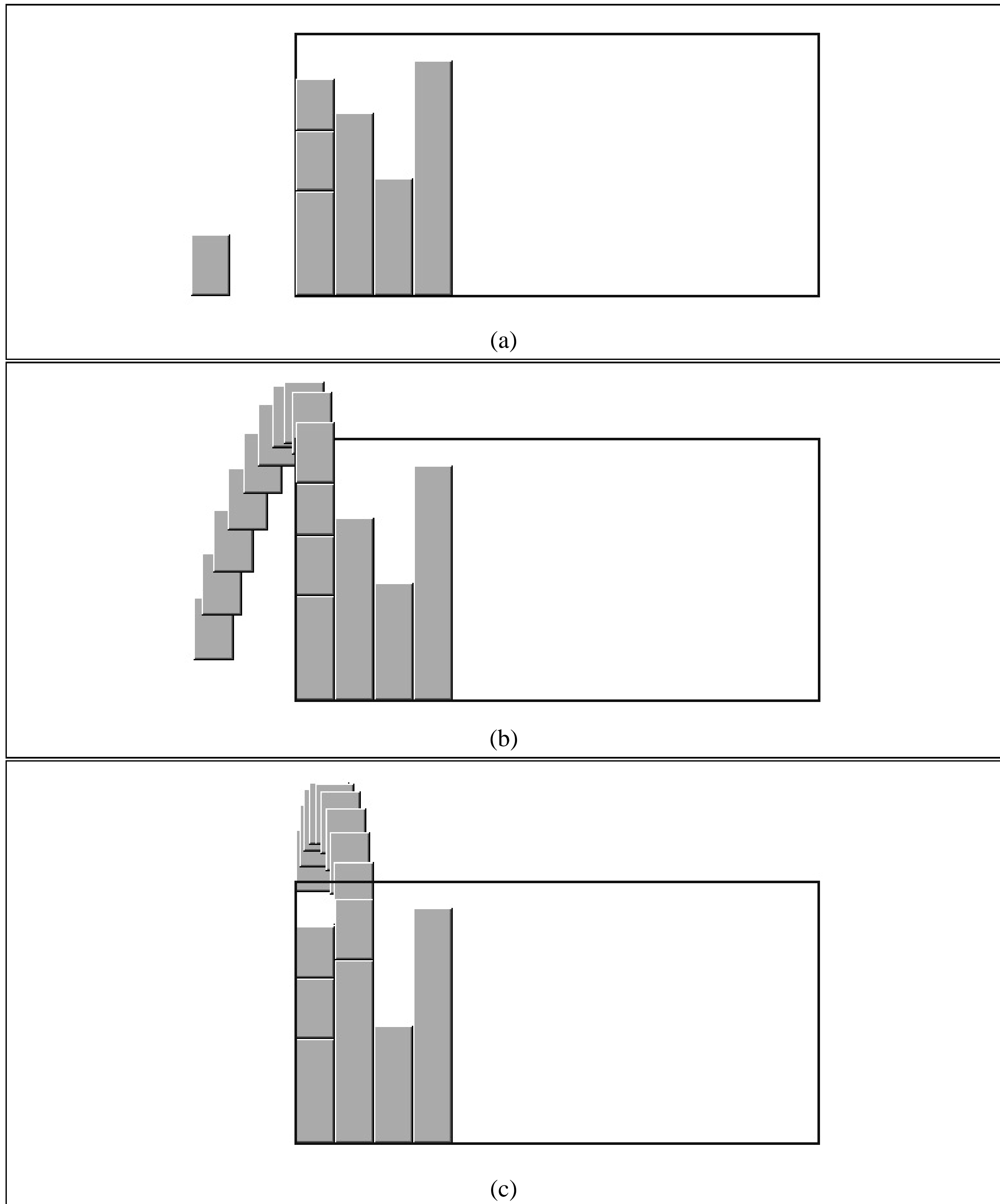


Figure 5.17: Bin-packing algorithm.

5.4.5 Finding a Minimum Spanning Tree

A spanning tree is a subset of an undirected graph that is acyclic and connects all the vertices. A minimum spanning tree is a spanning tree that has the minimum total weight of its edges. We have implemented and animated Kruskal's algorithm[26] for finding a minimum spanning tree from a given graph. Kruskal's algorithm consists of two phases. In the first phase, all edges are first removed and sorted, and in the second phase, edges are added one by one from lighter to heavier until all vertices are connected.

In this example, the user inputs the graph by drawing it with a MacDraw-like graphical editor. Figure 5.18 (a) shows the given graph. First, all edges are removed from the graph, and arranged vertically to the right (Figure 5.18 (b)). Thin edges in the figures mean indicate that they are not included in the graph. Second, the edges are sorted (Figures 5.18 (c) and (d)). Then, the edges are inserted into the graph from the one above (Figure 5.19 (e)), but any edges that make a cycle are excluded (Figures 5.19 (f) and (g)). The thick edges represent the edges that are included in the graph. Figure 5.19 (h) shows the final minimum spanning tree.

As TRIP2a has a special constraint solver for graph layout, only the data of nodes and edges are necessary for visualizing general undirected graphs such as that used in this animation.

5.5 Incorporating Event-Driven Animations

We have developed TRIP2a based on the bi-directional translation model that was described in the previous chapter. This tool can animate transitions of internal application data by specifying two mappings declaratively: (1) mapping between application data and its visual representation; and (2) mapping between operations executed in an application and corresponding motions in the visual representation.

However, as this model assumes that an animation is constructed by connecting short animations that depict the changes from one state to the immediately subsequent state, it is difficult to display extended motions that represent the change from one state to that some time later. To cope with this problem, we have incorporated event-driven animations into our framework, which enable us to specify an animation with just the start and end events.

This chapter first describes the basic model formally and the difficulties in showing extended motions. Then, we describe a slightly modified model that incorporates event-driven animation. Last, implementation of the model and its application to visualization of the execution of a concurrent program are described.

The implementation of the system is tailored to visualizing program executions and has been extended to handle 3D animations. A brief description of the system is attached as Appendix B.

5.5.1 The Basic Model

The model for creating animations described in the previous chapter assumes that an animation, A, is generated by inbetweening the sequence of frames $f_0, f_1, ..., f_n$ [9]. Animation A can therefore be written as:

$$A = I(f_0, f_1) + I(f_1, f_2) + ... + I(f_{n-2}, f_{n-1}) + I(f_{n-1}, f_n) \ , \tag{5.1}$$

where $I(f_i, f_{i+1})$ is a short animation generated by inbetweening frames f_i and f_{i+1}, and '+' indicates the concatenation of two animations. The frame f_i is a visualized picture translated from the

[9] Strictly speaking, we should refer to these as *keyframes*. However, as we do not refer to the frames generated by inbetweening keyframes, the word *frame* refers to *keyframe* in this chapter.

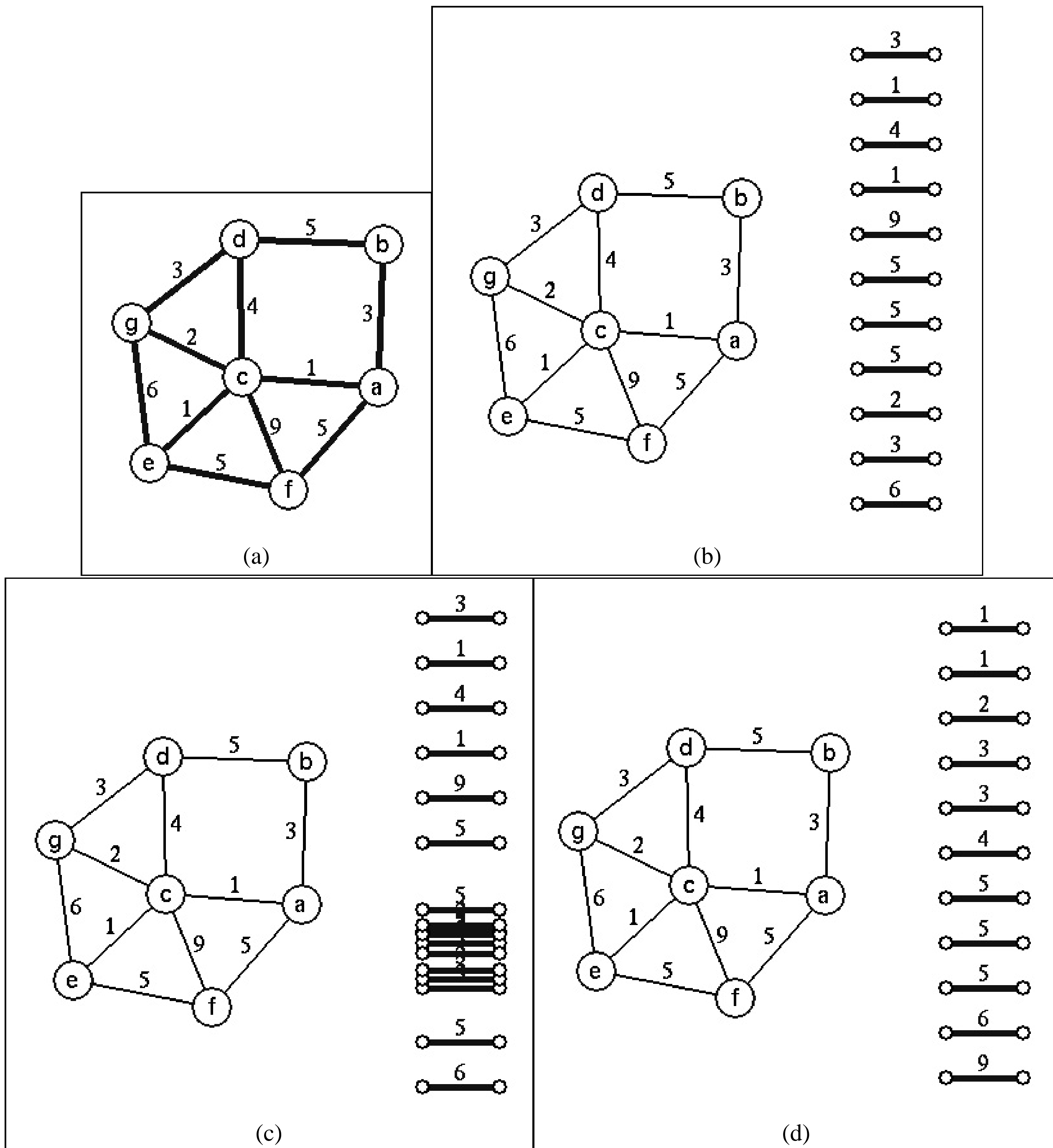


Figure 5.18: Minimum spanning tree algorithm (1).

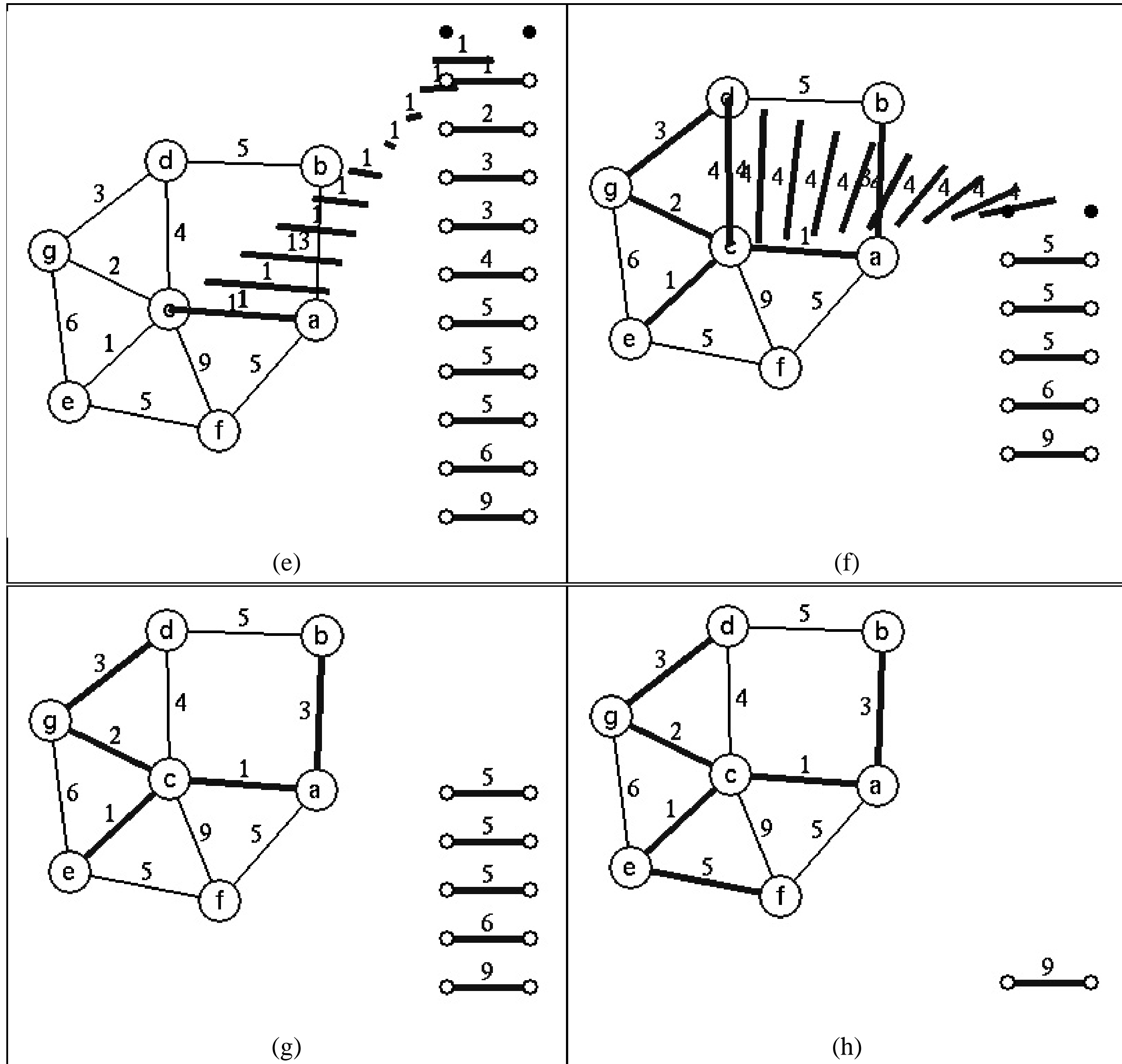


Figure 5.19: Minimum spanning tree algorithm (2).

internal state s_i by the visual mapping function $vm : S \to F$.

$$f_i = vm(s_i) \ . \tag{5.2}$$

This model assumes that the execution of a target application is a sequence of operations, op_i, that change the internal data of the application.

$$... \to s_i \xrightarrow{op_i} s_{i+1} \xrightarrow{op_{i+1}} s_{i+2} \to ... \tag{5.3}$$

As described above, the animation is a concatenated sequence of short animations $I(f_i, f_{i+1})$(Eqn 5.1). That is, each short animation $I(f_i, f_{i+1})$ represents the operation op_i.

5.5.2 The Problem

There are situations that cannot be represented well with the above model. As an example, let us consider the message-passing animation in which two nodes, A and B, send messages to each other. We represent these nodes as two boxes, and messages as spheres (Figure 5.20).

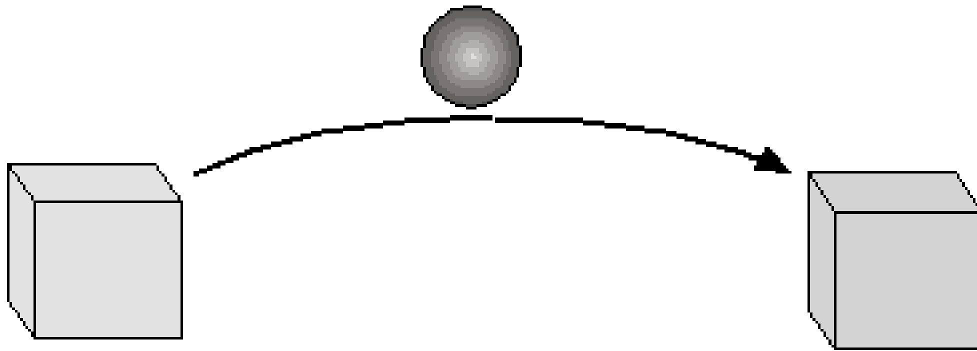


Figure 5.20: Sending a message.

With our model, the animation that shows a message m_1 being sent from node A to node B is constructed by concatenating three units of animation that interpolate four successive states. Each state is described by the state of the message. There are three types of state: (1) no message exists; (2) a message exists at node A; (3) a message exists at node B. The following are the four successive states that occur during message sending:

1. There are two nodes in the state.

```
a, b : node
```

2. The message m_1 is located at node A.

```
a, b : node
m1(a) : message
```

3. The message is located at node B.

```
a, b : node
m1(b) : message
```

4. There are two nodes in the state.

```
a, b : node
```

The three units of animation that interpolate the above states are:

$I(1,2)$ **:** The animation that shows the appearance of the message object.

$I(2,3)$ **:** The animation that shows the movement of the message object from node A to node B.

$I(3,4)$ **:** The animation that shows the disappearance of the message object.

The entire animation A_1 is a concatenation of the above animations, i.e., $A_1 = I(1,2) + I(2,3) + I(3,4)$.

Next, consider the case in which two messages m_1 and m_2 are sent from node A to node B. This is simple if the second message is sent after the arrival of the first at node B, because it is achieved by concatenating the above message-passing animation twice.

$$A_{2s} = A_1 + A_1 \tag{5.4}$$

However, the case in which the second message m_2 is sent before receipt of the first message is difficult to handle with the model. In this case, we cannot construct the animation by only concatenating the unit animations: $I(1,2)$, $I(2,3)$, $I(3,4)$.

To make the animation of this execution, these units of animation must be *overlapped*. However, this is difficult with our basic model, because it is necessary to determine the state when the message is still in transit and generate a keyframe by visualizing it. For example, when the second message m_2 starts moving, the first message m_1 is still in transit to node B. As the first message is at neither node A nor node B, the position of the message cannot be specified statically.

That is, in our model, (1) animations are created by concatenating short animation units, and (2) the programmer must specify the location and size of all objects at the start and end frames of each animation unit. It is difficult to satisfy these constraints if multiple objects start and stop moving asynchronously, because it is difficult to know the positions of moving objects as the system calculates them by interpolating frames.

5.5.3 Approach

To cope with this problem, we modified the model to enable specification of the ends of unit animations. The unit animations are constructed with respect to each object in keyframes. In the original model, it was assumed that unit animations of all objects are generated by interpolating the current and the next keyframe. The new model makes it possible to make unit animations of some objects by interpolating the current and distant keyframes. When interpolating distant keyframes, the in-between keyframes are ignored so that the programmer does not need to determine the positions of moving objects, which allows overlapping of unit animations.

More formally, the original and the new model are described as follows. Each frame f_i is a set of graphical objects.

$$f_i = \{obj_j^i | 0 \leq j \leq m\}$$

A unit animation of an object is written as $I_{obj}(obj_j^k, obj_j^l)$ which is a motion from obj_j^k (the object obj_j in the frame f_k) to obj_j^l. The animation in the original model is written as follows.

$$I(f_i, f_{i+1}) = \bigoplus_j I_{obj}(obj_j^i, obj_j^{i+1}) \quad (5.5)$$

$$A = \sum_{i=0}^{n-1} I(f_i, f_{i+1}) \,. \quad (5.6)$$

In the above equation, '$\oplus$' means parallel concatenation of motions of objects in which element animations may overlap, and 'Σ' means sequential concatenation of unit animations. On the other hand, the animation in the new model is written as follows.

$$A_{new} = \bigoplus_{j,k} I_{obj}(obj_j^{e_j(k)}, obj_j^{e_j(k+1)}) \,. \quad (5.7)$$

Here, $e_j(k)$ is the kth number of a monotone increasing subsequence of $0...n$. Figure 5.21 shows both the original model and the new model. To differentiate the new type of unit animations, we

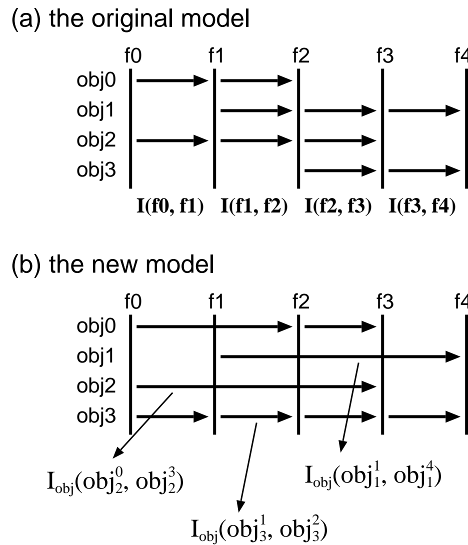


Figure 5.21: The original & new models.

call $I_{obj}(obj_j^k, obj_j^l)$ *long actions* if $k+1 < l$. On the other hand, we call $I_{obj}(obj_j^k, obj_j^{k+1})$ *instant*

actions. For example, in Figure 5.21, $I_{obj}(obj_2^0, obj_2^3)$ is a long action, and $I_{obj}(obj_3^1, obj_3^2)$ is an instant action. The actions in the original model (Figure 5.21(a)) are all instant actions.

To make the animation $I_{obj}(obj_2^0, obj_2^3)$, only the keyframes f_0 and f_3 are required, and f_1 and f_2 are not necessary. Similarly, to make $I_{obj}(obj_1^1, obj_1^4)$, the keyframes f_1 and f_4 are required, and f_2 and f_3 are not necessary. Therefore, the programmer does not need to specify the positions of moving objects. For example, the frame f_1 is specified only by the positions of obj_1 and obj_3, and the frame f_2 is specified only by the positions of obj_0 and obj_3. This enables the overlapping of unit animations, such as $I_{obj}(obj_2^0, obj_2^3)$ and $I_{obj}(obj_1^1, obj_1^4)$.

5.5.4 Implementation

Based on the new model described in the previous section, we have built the TRIP2a/3D system to handle long actions. This section describes the implementation details of the new TRIP2a/3D system.

Specifying Long Actions

A long action is specified by a pair of transitional operations. Usually, each operation corresponds to an event during the execution of an application. The programmer modifies the program so that it writes out the events and the current state together to a log file when such events occur during its execution.

Events are treated as abstract operations on ASR. Similar to abstract operations, the programmer defines events as terms, but each must have an ID number. The ID number of an event is necessary to allow searching for a pair of events. Two events with the same ID are considered to be a pair of events that represent a long action. The event data are translated to special transitional operations of type `move/2`, which indicate long actions. The translation from events to transitional operations is specified by mapping rules written by the programmer.

As an example, consider the message-passing animation example described above. In this case, sending a message corresponds to the start event of a long action, and receiving a message corresponds to the end event. First, the programmer defines these events: `snd(Obj, From, Id)` and `rcv(Obj, To, Id)`. This means that the message `Obj` is sent from `From` to `To`. The third variable, `Id`, is the ID number of the pair of events.

Then, the programmer defines mapping rules that map these two events to special transitional operations. These mapping rules are written as follows:

```
rule(snd(Obj, From, Id), Result) :-
   Result = [move(Obj, [from(From,Id)])].
rule(rcv(Obj, To, Id), Result) :-
   Result = [move(Obj, [to(To,Id)])].
```

Thus, two types of events, `snd(Obj, From, Id)` and `rcv(Obj, To, Id)`, are mapped to the transitional operations `move(Obj, [from(From,Id)])` and `move(Obj, [to(To,Id)])`, respectively.

The variables `Obj`, `From`, `To` in the arguments `move/2` are IDs of graphical objects. The two `moves` that have the same ID[10] are regarded as a pair, and are thought to represent the long action in which `Obj` moves from the position of `From` to the position of `To`.

[10] This ID is unique to this type of transitional operation, and is not concerned with the IDs of graphical objects.

Generating an Animation for a Long Action

Chapter 5 described how animation is generated by interpolating pairs of keyframes that are translated from application data. In our new model, animations that represent long actions are generated in the same way, but this leads to a problem in implementing the system. In the previous model, the two keyframes to be interpolated were adjacent, and a unit of animation was generated by interpolating two adjacent frames. In the new model, the two keyframes that correspond to the start and end events may be distant. Even when the system finds a frame that corresponds to the start frame of a long action, it cannot know when the end frame will appear in the execution log. As both events are necessary to generate an animation for a long action, the system will have to search for another event for the rest of the log file once one of them is found.

There is no problem if the animation is generated after the execution is finished, because the entire log can be obtained and searched. However, if the animation must be displayed parallel to the execution of the target application, the start of the animation for a long action must be delayed until the end event occurs. Currently, we only generate animations after the execution has finished.

Architecture

Figure 5.22 shows the architecture of TRIP2a/3D. The programmer creates the hatched area in Figure 5.22 for each target application. The programmer first instruments the application to output execution logs. Unlike TRIP2a, TRIP2a/3D reads only from text log files. The application must output its internal data and events into a text file, and then TRIP2a/3D reads and visualizes them. When it is difficult to output data directly in the format of ASR terms, especially when they are distributed applications, the programmer must write a program to collect log files from each executed application and compile them into the ASR data file.

The programmer then writes visual mapping rules, and builds a *mapper* module for each type of execution log. Visual mapping rules are compiled and linked with other libraries. They are written in KLIC [23], a concurrent logic programming language. The resulting module reads the input ASR data and translates them into animation scripts for the viewers.

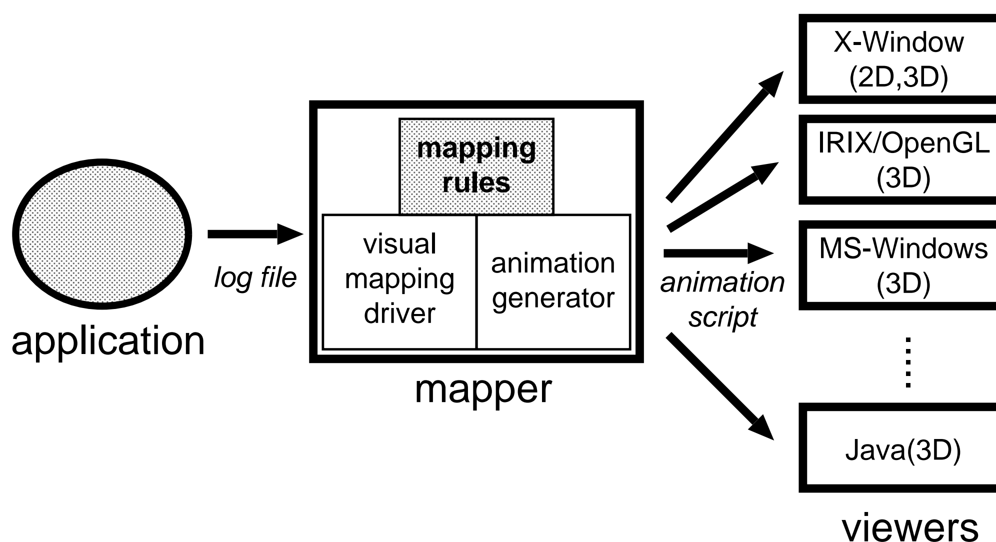


Figure 5.22: Architecture of TRIP2a/3D.

The major difference from the implementation of TRIP2a is that the viewers and the visual

mapping module are separate, and several viewers are provided for different platforms. It is also possible to view animations in different ways, such as two-dimensionally or three-dimensionally, by changing the viewer. The three-dimensional viewer is implemented with Open Inventor[126] and OpenGL, and the two-dimensional viewer is implemented with Tcl/Tk and STk. We have also implemented the viewer with Java using Java3D. The multiple viewers can be easily implemented, because we defined a simple *animation script*, which is a series of low-level operations on graphical objects such as move objects and change size. The viewers are interpreters that interpret the animation scripts. Their syntax is described in Appendix B.

Figure 5.23 shows a screenshot of the viewer implemented with Java. The viewer has two modes. In animation mode, the animation is displayed in the window. By pressing the button at the lower-left corner of the window, the user can switch the viewer to the step mode. In this mode, the user can freely wind the animation forward or backward using the slider. The window displays the still image at the moment corresponding to the slider position. The 3D viewing position can be changed with a mouse and a keyboard. The user can rotate the displayed objects by dragging on the window, and can shift the view or zoom-in/out by using the cursor keys and Page-Up/Down keys, respectively.

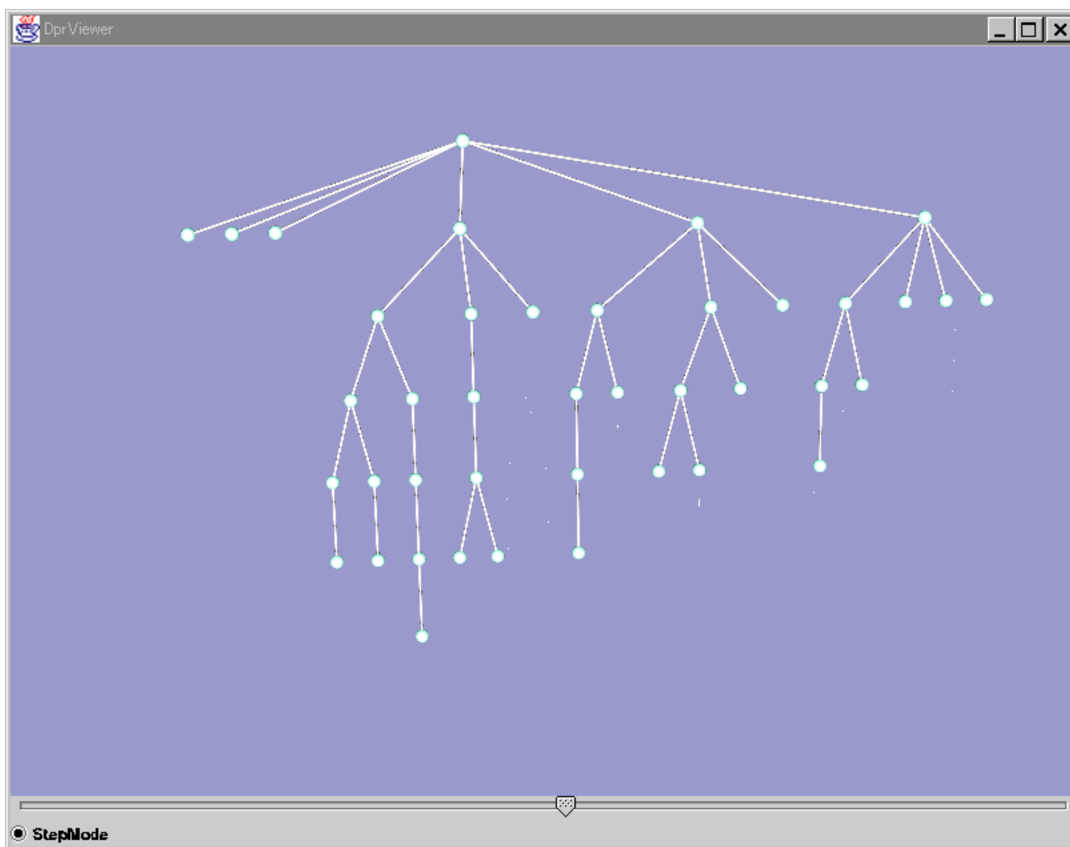


Figure 5.23: Screenshot of Java-TRIP2a/3D.

5.5.5 An Example

The N-Queen Problem

Figure 5.24 shows a search tree for the N-Queen (N=6) problem. Each branch represents a choice of position where a queen can be placed. In this search, queens are placed from the first to the last row in order. Only one queen can be placed in each row, so there are only six choices. However, the program cuts branches when some of the choices conflict with the queens already in position. If there is no choice, the node cannot extend a branch.

This animation shows two types of action. The first type of action is the extension of branches. When the N-Queen program has found the position choices for the next row, the animation shows the emergence of new child nodes. For example, when the program has found that there are three choices of position, the animation shows the creation of three new children (branches).

Another type of action is the sending of messages to the root node. A message is sent when the program has succeeded in placing six queens on the board. In this case, messages are sent from the node at a depth of six. A message is also sent when the program has failed to find a choice of position in the next row. In this case, the program cannot extend any more branches, and the depth of such a node is less than six.

In the animation, the messages are represented as flying circles, which fly from the source node to the root node. The color of nodes in the search tree represents the machine on which the search is executed. For example, two machines are used for the search shown in Figure 5.24. The left and the right halves of the tree are executed on different machines. The light green nodes on the left part of the tree are executed on one machine (A), and the other brown nodes are executed on another (B).

The execution is started on machine A. The first six choices are divided into two parts, and half are sent to machine B. The execution on machine A proceeds faster than that on machine B (Figure 5.24). In addition, the message from the last node arrives immediately at the root node. On the other hand, many messages have been sent from machine B, but have yet to arrive at the root node (Figure 5.24), because the left part of the tree is executed on a different machine from the root node. This is represented by many flying circles in the animation.

Figures 5.25 and Figure 5.26 show a program that solves the N-Queen problem[11]. The program runs on several machines in parallel. The log data output from the program are shown in Figure 5.27. The data are converted to ASR by the special converter. The resulting ASR are shown in Figure 5.28. The visual mapping rule for displaying a search tree is presented in Figure 5.29.

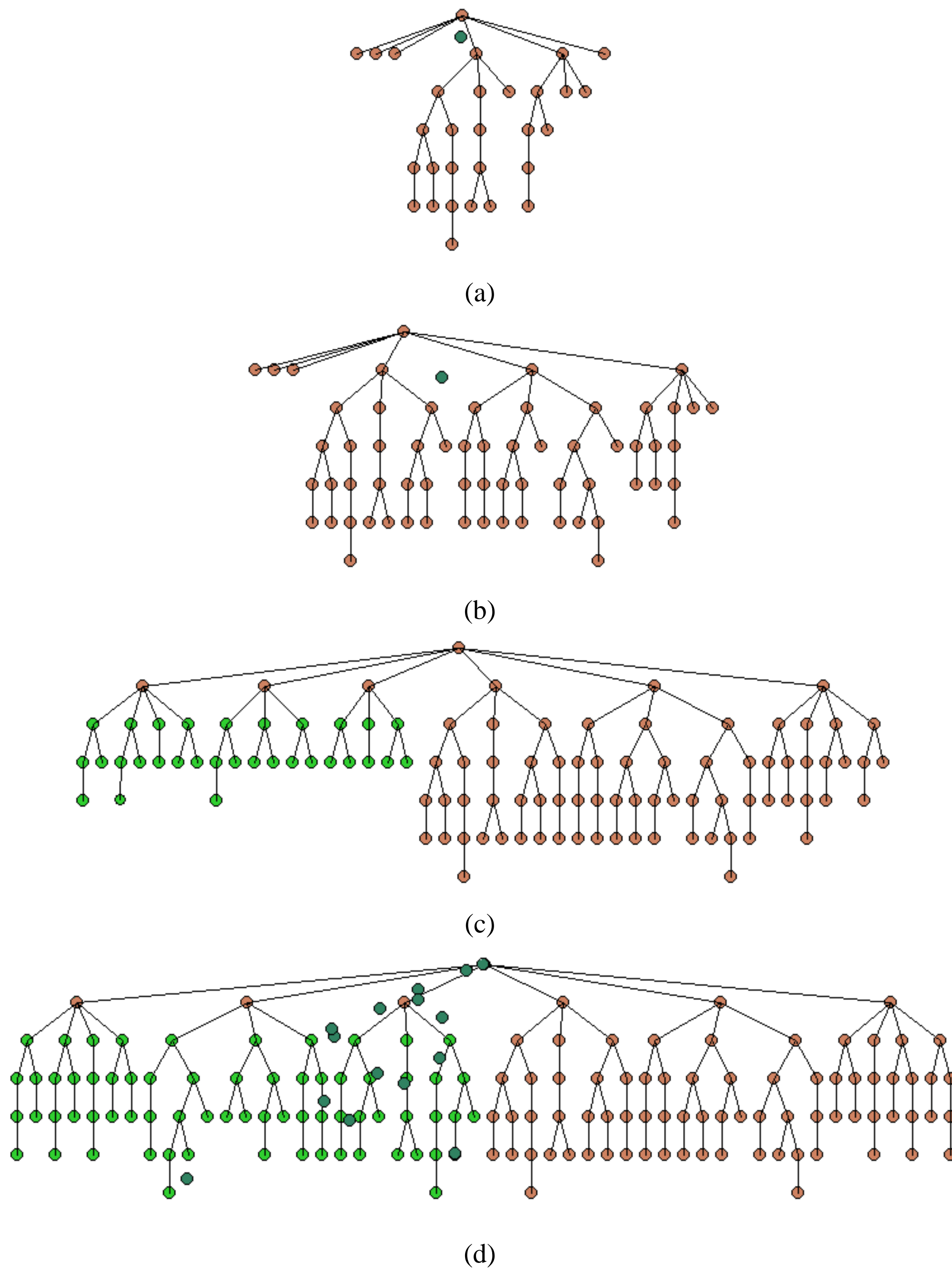
5.5.6 Summary

We have described the problem of parallel program animation in our previous model and the solution. In our previous model, it was difficult to overlap unit animations. This problem is alleviated in our new model by allowing programmers to specify the start and end events of unit animations. We have also described the TRIP2a/3D system based on the new model and an example animation created with TRIP2a/3D.

5.6 Conclusions

We have proposed the extended bi-directional translation model for animations, and have implemented a prototype system based on this model. Animations can be created by providing visual and

[11]This program was written in Schematic[122].



(a)

(b)

(c)

(d)

Figure 5.24: Screenshot of an animation that shows a search tree for the N-Queen problem.

```
;;;
;;; N-Queen Probelm
;;;
(module schematic-user)
(include "schematic2.sch")
(include "counter.sch")
(include "logger.sch")

(definep (n-queen-sub c j N M soremade l)
  (cond ((> j N)
          (begin
             (now (output-log-with-time l "~s ~s ~s () s̈end~̈%"
                (list *pe* j soremade)))
             (add! c 1)
             (now (output-log-with-time l "~s ~s ~s () s̈ent~̈%"
                (list *pe* j soremade)))
             1))
        ((= j 1)
          (let ((candidates (safe-place-list 1 1 N soremade)))
            (if (null? candidates)
                (begin
                  (now (output-log-with-time l "~s ~s ~s ~s s̈end~̈%"
                        (list *pe* j soremade candidates)))
                  (add! c 1)
                  (now (output-log-with-time l "~s ~s ~s ~s s̈ent~̈%"
                        (list *pe* j soremade candidates))))
                (now (output-log-with-time l "~s ~s ~s ~s~%"
                        (list *pe* j soremade candidates))))
            (psum (map (plambda (i)
                         (let ((processor
                                  (quotient
                                     (- N i)
                                     (inexact->exact (ceiling (/ N M))))))
                           (future (n-queen-sub c 2 N M (cons i sore-
made) l)
                                  :on processor)))
                           candidates))))
        ((or (= j 2) (= j 3))
          (let ((candidates (safe-place-list 1 j N soremade)))
            (if (null? candidates)
                (begin
                  (now (output-log-with-time l "~s ~s ~s ~s s̈end~̈%"
                        (list *pe* j soremade candidates)))
                  (add! c 1)
                  (now (output-log-with-time l "~s ~s ~s ~s s̈ent~̈%"
                        (list *pe* j soremade candidates))))
                (now (output-log-with-time l "~s ~s ~s ~s~%"
                        (list *pe* j soremade candidates))))
            (psum (map (plambda (i)
                         (future (n-queen-sub c (+ j 1) N M (cons i sore-
made) l)))
                         candidates))))
        (else
          (let ((candidates (safe-place-list 1 j N soremade)))
            (if (null? candidates)
                (begin
                  (now (output-log-with-time l "~s ~s ~s ~s s̈end~̈%"
                        (list *pe* j soremade candidates)))
                  (add! c 1)
                  (now (output-log-with-time l "~s ~s ~s ~s s̈ent~̈%"
                        (list *pe* j soremade candidates))))
                (now (output-log-with-time l "~s ~s ~s ~s~%"
                        (list *pe* j soremade candidates))))
            (sum (map (plambda (i)
                           (n-queen-sub c (+ j 1) N M (cons i soremade) l))
                      candidates))))))
```

Figure 5.25: The target parallel N-queen-solving program (1).

```
(definep (safe-place? i j y soremade)
  (if (null? soremade) #t
      (let ((x (car soremade)))
        (cond ((= i x) #f)
              ((= (- i j) (- x y)) #f)
              ((= (+ i j) (+ x y)) #f)
              (else
               (safe-place? i j (- y 1) (cdr soremade)))))))

(definep (safe-place-list i j N soremade)
  (if (> i N) '()
      (if (safe-place? i j (- j 1) soremade)
          (cons i (safe-place-list (+ i 1) j N soremade))
          (safe-place-list (+ i 1) j N soremade))))

(definep (psum list)
  (if (null? list) 0 (+ (touch (car list)) (psum (cdr list)))))

(definep (sum list)
  (if (null? list) 0 (+ (car list) (sum (cdr list)))))

(definep (n-queen N)
  (let* ((c  (counter 0))
         (t0 (current-time))
         (n  (n-queen-sub c 1 N *npe* '() (logger)))
         (t1 (current-time))
         (m  (get-value c)))
    (format #t "~s ~s : ~s msec~%" n m (- t1 t0))))

(definep (schematic-main args)
  (n-queen (string->number (cadr args)))
  (newline))

;;;
;;; counter.sc---Simple Counter class
;;;

(define-generic (add! self x))
(define-generic (get-value self))

(define-class counter ()
  value)

(define-method! counter (add! self x)
  (become value :value (+ x value)))

(define-method counter (get-value self)
  value)

;;;
;;; an object for logging
;;;

(define-class logger ())

(define-generic (output-log-with-time self log-format args))
(define-generic (output-log self log-format args))

(define-method logger (output-log-with-time self log-format args)
   (format #t "~s  " (current-time))
   (apply format `(#t ,log-format ,@args)))

(define-method logger (output-log self log-format args)
   (apply format `(#t ,log-format ,@args)))
```

Figure 5.26: The target parallel N-queen-solving program (2).

```
30659177  0 1  () (1 2 3 4 5 6)
30659179  0 2  (4) (1 2 6)
30659179  0 3  (1 4) (3 5)
30659179  0 4  (3 1 4) (5 6)
30659180  0 5  (5 3 1 4) (2)
30659180  0 6  (2 5 3 1 4) ()
30659180  0 5  (6 3 1 4) (2)
30659180  0 6  (2 6 3 1 4) ()
30659181  1 2  (1) (3 4 5 6)
30659181  1 2  (2) (4 5 6)
30659182  1 2  (3) (1 5 6)
30659183  1 3  (3 1) (5 6)
30659184  0 2  (5) (1 2 3)
30659184  0 3  (1 5) (4 6)
30659185  0 4  (4 1 5) (6)
30659185  0 5  (6 4 1 5) (3)
30659185  0 6  (3 6 4 1 5) ()
 ...omitted...
```

Figure 5.27: An example of log data.

```
node(t0,red).
node(t1,red).
node(t2,red).
node(t3,red).
node(t4,red).
node(t5,red).
node(t6,red).
trees([tree(t0,[t1,t2,t3,t4,t5,t6])
]).
end_of_state.
node(t0,red).
node(t1,red).
node(t2,red).
node(t3,red).
node(t4,red).
node(t10,red).
node(t16,red).
node(t40,red).
node(t5,red).
node(t6,red).
trees([tree(t0,[t1,t2,t3,t4,t5,t6])
,tree(t4,[t10,t16,t40])
]).
end_of_state.
node(t0,red).
node(t1,red).
node(t2,red).
node(t3,red).
node(t4,red).
node(t10,red).
node(t118,red).
node(t190,red).
node(t16,red).
node(t40,red).
node(t5,red).
node(t6,red).
trees([tree(t0,[t1,t2,t3,t4,t5,t6])
,tree(t4,[t10,t16,t40])
,tree(t10,[t118,t190])
]).
end_of_state.
...omitted...
```

Figure 5.28: The ASR data for N-queen animation.

```
:- module vmr.

rule(default, Result) :- true | Result = [].

rule(node(X), Result) :- true |
        Result = [sphere(X, 5, [material([ambient(0.2, 0.5, 0.4)])])].
rule(node(X, red), Result) :- true |
        Result = [sphere(X, 5, [material([ambient(0.8, 0.5, 0.4)])])].
rule(node(X, blue), Result) :- true |
        Result = [sphere(X, 5, [material([ambient(0.2, 0.2, 0.8)])])].
rule(node(X, green), Result) :- true |
        Result = [sphere(X, 5, [material([ambient(0.2, 0.8, 0.2)])])].
rule(send(Mess, Id, From), Result) :-
        Result = [move(Mess, [from(From, Id)])].
rule(sent(Mess, Id, To), Result) :-
        Result = [move(Mess, [to(t0, Id)])].

rule(trees(X), Result) :- true |
        treesmap(X, Res1),
        layout_leaf(X, Res2),
        generic:new(merge, {Res1, Res2}, Result).

%%%---------------------------------------

treesmap([], Result) :- true | Result = [].
treesmap([tree(X, L)|T], Result) :- true |
        L = [H|_],
        treesmap(T, RestResult),
        connections(X, L, Connections),
        TmpResult = [x_parallel(L, []),
                    x_average(X, L, []),
                    z_relative([X, H], 0, []),
                    y_relative([X, H], -30, [])],
        generic:new(merge, {Connections, RestResult, TmpResult}, Result).


connections(X, [], Connections) :- true | Connections = [].
connections(X, [H|T], Connections) :- true |
        Connections = [line(l(X, H), 1, []), con-
nect(l(X, H), X, H, [])|Rest],
        connections(X, T, Rest).

layout_leaf(List, Result) :- true |
        List = [tree(X, L)|Rest],
        makeLeafList([X], List, Leafs),
        trip:flat(Leafs, FlatLeafs),
        makeLeafConstraints(FlatLeafs, Relatives),
        %Result = [place(X, 0, 0, 0, []), x_relative(FlatLeafs, 15, [])].
        Result = [place(X, 0, 0, 0, [])|Relatives].

makeLeafConstraints([_], R) :- true | R = [].
makeLeafConstraints([X, Y|T], R) :- true |
        R = [x_relative([X, Y], 15, [])|R1],
        makeLeafConstraints([Y|T], R1).


makeLeafList([], Trees, Result) :- true | Result = [].
makeLeafList([Node|Rest], Trees, Result) :- true |
        searchTree(Node, Trees, Tree),
        subMakeLeafList(Node, Tree, Trees, R1),
        makeLeafList(Rest, Trees, R2),
        generic:new(merge, {R1, R2}, Result).

subMakeLeafList(Node, nil, Trees, R) :- true | R = [Node].
otherwise.
subMakeLeafList(Node, Tree, Trees, R) :- true |
        Tree = tree(X, L),
        makeLeafList(L, Trees, R).

searchTree(Node, [], Tree) :- true | Tree = nil.
searchTree(Node, [tree(Node, L)|Rest], Tree) :- true | Tree = tree(Node, L).
otherwise.
searchTree(Node, [T|Rest], Tree) :- true | searchTree(Node, Rest, Tree).
```

Figure 5.29: The visual mapping rule set for the N-queen animation.

transitional mapping rules, and by annotating an application program so that the data and the operation of the application are passed to the animation system. We have applied this model to several algorithms, such as sorting algorithms, the tower of Hanoi, and Kruskal's algorithm for finding a minimum spanning tree. Our experience has shown that animations can be created quite effortlessly.

In future studies, we plan to provide a way to specify animations by demonstration using the techniques of TRIP3/IMAGE[88, 87] described in Appendix A.

Chapter 6

Visualizing and Browsing Constraints in Visualization Rules

The visualization rules for TRIP systems are sets of mapping rules that map application source data (ASR data) to the high-level representation of the target picture (VSR data). The VSR data consist of geometric graphical objects and the geometric constraints among them. If the visualization rules have bugs, the TRIP system cannot solve the constraints in the VSR data properly and cannot generate a target picture.

To support debugging of TRIP visualization rules, we built a prototype tool for browsing the constraint system in TRIP systems' visualization rules. It has two views that show the constraint system. In one view, the tool visualizes a constraint system as a three-dimensional graph structure, which shows the overall structure of the constraint system. The other view shows the target pictures, and animates them so that the degrees of freedom of graphical objects in the constraint system are presented to the user. The HiRise constraint solver is incorporated into this tool to deal with under-constrained systems. In addition, the cartoon technique of deforming graphical objects is utilized to depict whether a graphical object has a degree of freedom.

6.1 Introduction

Constraints are widely applied for various problems in GUIs, such as drawing editors, visualization systems, and GUI development tools. In these systems, geometric constraints are used to specify relations among graphical objects on the screen. In Amulet[89] and SubArctic[107], more general constraints among variables are used.

The most important merit of constraints is their declarativity. By using constraints, the programmer does not need to write a procedure to achieve an intended goal, as the desired results can be written directly. Constraint solvers automatically satisfy constraints by substituting appropriate values into constrained variables.

However, a constraint solver does not necessarily provide primitive constraints that directly achieve the programmer's intended result. In the case of drawing editors, it may be possible to provide required constraints to decorate figures interactively. In general, to specify visualization or to program with constraints, it is necessary to combine a number of primitive constraints to obtain the desired result. Therefore, there are various difficulties in programming with constraints. For example, there are cases in which each primitive constraint seems appropriate but the combined constraint system does not work well. Such constraint systems are difficult to debug.

One reason for the difficulties in debugging constraints is that representing constraints is difficult. Constraints are highly abstract concepts that check whether they are satisfied or not. One way to represent them is to show constrained objects that satisfy constraints. However, there are multiple states that satisfy a constraint system, which should be represented together to aid in comprehension.

The focus of this chapter is on the help of the debugging of a constraint system contained in a set of TRIP-system visual mapping rules[67, 121, 119]. These rules specify how source data should be visualized into the target picture. They are declarative rules that map application data into graphical objects and relations (constraints). TRIP systems interpret graphical objects and relations as a picture, and show them to the user. The visual mapping rules are difficult to debug, and originally TRIP systems had little support for debugging rules.

In this chapter, two approaches to visualizing mapping rules are described. Their purpose is to help the programmer to grasp the overall structure of constraints in rules. The first approach is to visualize them as an undirected three-dimensional graph structure. Constraints and constrained objects are the nodes. Edges connect constraints and constrained objects. Exploring the structure of the graph can be achieved by changing their layout in several ways so that the structure of the focused sub-graph becomes apparent.

The second approach is to represent constraint systems using animation. By using animation, dynamic aspects of constraint systems can be represented.

- Degrees of freedom of objects are represented by animation. Constraining objects decreases their degree of freedom.

- Whether an attribute has any degree of freedom can be represented by its display. An attribute with a degree of freedom is animated so that its various valid values are shown to the user. A constrained attribute is animated as appearing to satisfy the constraints.

- An attribute that has no degrees of freedom, i.e., an attribute that has a determined value, is animated (jerked or twitched) as if it is trying to change its value but cannot. Cartoon animation techniques are used to represent this situation.

- Attributes that are related are animated together. That is, attributes that have no relation to each other should not be animated together.

This chapter is organized as follows: Section 2 describes the first approach to visualize constraint systems, and the tool for drawing three-dimensional graphs of constraints. In Section 3, we present a technique for animating the degrees of freedom in the constraint system. Section 4 discusses related work, and our conclusions are presented in Section 5.

6.2 Browsing Three-Dimensional Constraint Graphs

6.2.1 Basic Representation

Visual mapping rules are textual programs that map Abstract Structure Representation (ASR) data to Visual Structure Representation(VSR) data[1]. ASR consists of a set of ground compound terms which represents the structure of abstract data, and is used as a proxy for the application data in the model. VSR is a high-level representation of a picture, and consists of a set of graphical objects and graphical constraints. By using these two representations, programmers can specify the mapping between application data and a picture independently of each specific representation.

[1]These terms are described in the bi-directional translation model[67, 121, 119]. See them for more details.

Originally, TRIP systems did not have any support for writing mapping rules. To debug a mapping rule set, the programmer had to check the generated picture. However, when there are problems in the visual mapping rules, for example, if some rules are missing or conflict with each other, the TRIP system cannot generate a picture. In such cases, the programmer can only know that there are some problems in their rules, and can only view the text VSR data generated by the mapping rules.

To improve this situation, we have implemented a tool for visualizing VSR data as a three-dimensional graph structure. This tool has two windows. One window displays a resulting picture[2], and the other shows the corresponding three-dimensional constraint graph. Figure 6.1 shows an example snapshot of the constraint graph generated by our system. It represents the structure of the constraints in the VSR data shown in Figure 6.2. Figure 6.3 shows the target picture generated from the VSR data. The user browses and compares the target picture and the constraint graph with the viewer. The user can freely rotate and zoom in/out of the visualized picture and the 3D graph with a mouse.

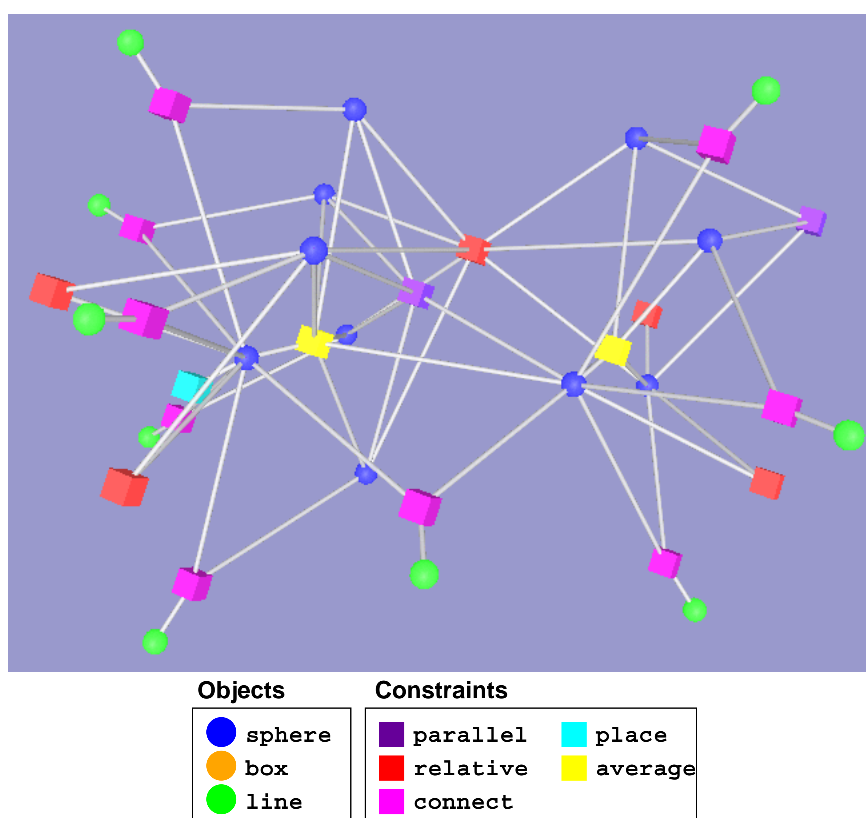


Figure 6.1: The constraint graph corresponding to the VSR data in Figure 6.2.

Two types of VSR data are depicted in Figure 6.1. One type consists of geometric primitive objects, such as boxes, spheres, and lines. These are represented as sphere nodes in the visualized constraint graph. The color of a node shows the type of geometric object. For example, the green spheres represent line objects, and the blue spheres represent sphere objects. The other type of VSR data consists of geometric constraints. Geometric constraints are represented as box nodes in the constraint graph. As with geometric objects, the colors of these objects represent the types of

[2]Pictorial Representation (PR) in the bi-directional translation model.

```
% The nodes in a tree are repseresented as spheres.
% sphe ID Radius Color
sphe 0 5 ambi 0.8 0.5 0.4
sphe 1 5 ambi 0.8 0.5 0.4
    ... Omitted ...
sphe 9 5 ambi 0.8 0.5 0.4
% Constraints that layout the nodes in a tree.
% para {x,y,z} ObjList — The objects in ObjList are arranged in a direction parallel to the {x,y,z}-axis.
% aver {x,y,z} ObjList — Obj1 is put at the average position of the objects in ObjList.
% rela {x,y,z} D ObjList — The distance between each successive pair of objects in ObjList in the direction of {x,y,z}-coordinate is D.
para x 1 2 3 4 8 9
aver x 0 1 2 3 4 8 9
rela z 0 0 1
rela y -30 0 1
para x 5 6 7
aver x 4 5 6 7
rela z 0 4 5
rela y -30 4 5
% Line objects and connect constraints.
% line ID Thickness
% conn Line Node1 Node2 — Node1 is connected to Node2 via Line.
line 10 1
conn 10 4 5
line 11 1
conn 11 4 6
    ... Omitted ...
line 18 1
conn 18 0 9
% The node t0 is placed at (0, 0, 0).
plac 0 1 2 3
% The distance between each pair of adjacent leaves in a tree is 15.
rela x 15 1 2 3 5 6 7 8 9
```

Figure 6.2: VSR data for a tree.

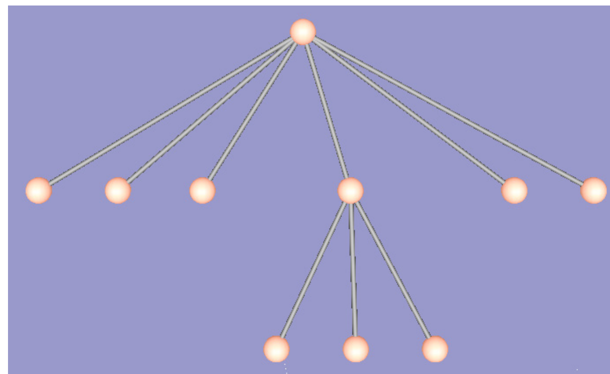


Figure 6.3: The resulting picture corresponding to Figure 6.2.

constraints.

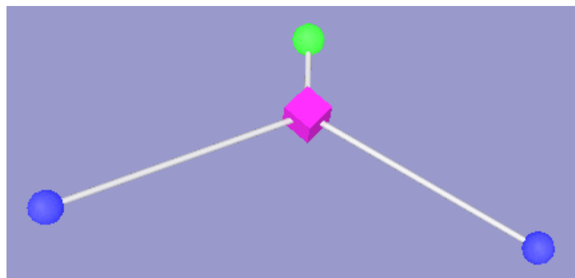


Figure 6.4: A connect constraint (purple box).

A box node is connected to spheres, i.e., a constraint node is connected to object nodes that are constrained by the constraint node. For example, the purple box in Figure 6.4 represents a connect constraint. It has three edges; one edge is connected to a "line" object (the green sphere), and the other two edges are connected to a "sphere" object (the blue sphere). This constraint arranges these three objects so that a line object connects two sphere objects. Figure 6.1 contains nine connect constraints, which correspond to nine edges of the tree shown in Figure 6.3.

Clicking a graphical object in the taget picture window highlights its correspondent graphical object node (sphere node) in the 3D constraint graph and vice versa, which helps users to find the correspondence between graphical objects in the picture and the object nodes in the 3D constraint graph. Clicking a constraint node (box node) in the constraint graph highlights constrained objects in the target picture window. For example, Figure 6.5 shows four nodes highlighted. they are constrained by an `average` constraint which constrains an upper node should be positioned at the average position (in x-coordinate) of the other three nodes at the bottom. Because the user clicked at the corresponding `average` constraint node in the constraint graph, they are highlighted.

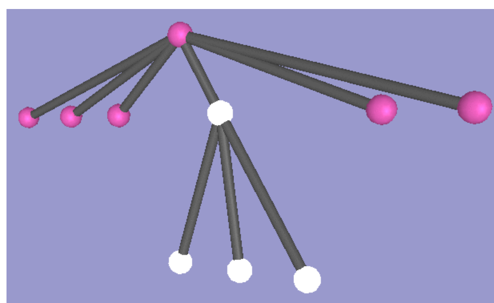


Figure 6.5: Nodes constrained by the clicked constraint are highlighted.

The merit of the constraint graph representation is that it can represent the structure of VSR more directly than text representation. It is helpful when debugging visual mapping rule sets to look at the structure of VSR translated from the application data with the rule. For example, the constraint graph shown in Figure 6.6 represents a constraint system that arranges ten boxes. The constraints used here are `relative` constraints that constrain the distance (in one dimension) between each object in their arguments. Each object has three degrees of freedom corresponding to the x-, y-, and z-coordinates. Therefore, each object should be constrained by three constraints. In Figure 6.6, the spheres that represent graphical objects are connected via three constraints, which are represented by edges and boxes, from which the user can understand that the objects are well constrained.

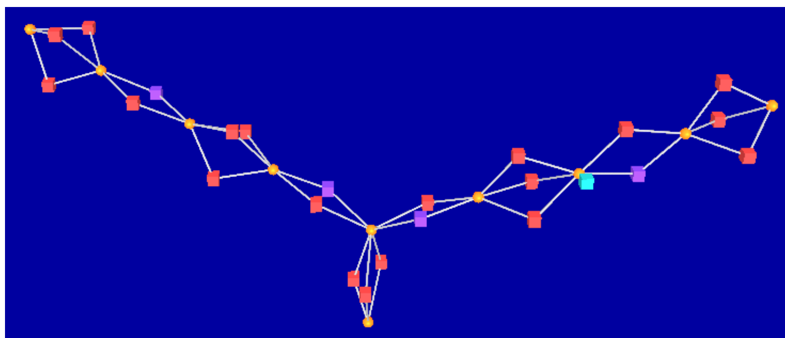


Figure 6.6: A constraint graph with regular structure.

Our tool can layout graphs three-dimensionally or two-dimensionally. Three-dimensional representation has more degree of freedom so that it is relatively faster in our system to lay out graphs three-dimensionally, especially for large and complex graphs. Overlapping of edges can be avoided easily in three-dimensional representation. However, since the user usually views their projected images in two-dimensional display, in order to understand the structure of the graphs, the user must rotate and view them from various directions. It is hard to avoid overlapping of edges in the two-dimensional layout of graphs. It is also complex for the user to manually change the layout of nodes in the graph to explore the structure of constraint graphs.

6.2.2 Changing Layout to Explore Constraint Graphs

In order to help the user to modify the layout of constraint graphs to explore the often tangled constraint graphs like the graph in Figure 6.1, we provide two ways to simplify the constraint graphs. One way is to lengthen the edges of the selected constraints, which makes them unfocused and simplifies the layout of the constraint graph. Figure 6.8 shows a constraint graph simplified by stretching the eight edges connected from a `relative` constraint of the graph shown in Figure 6.7. Since this constraint are related to many objects, it pulls other nodes to its neighbor position, which causes the structure of the graph more complex. Stretching the edges of a constraint makes the layout of the graph as if it is removed from the graph, but the connections still remain as pale translucent lines. By stretching the edges of two more `average` constraints (yellow boxes), the graph becomes much simpler (Figure 6.9). We can see that the layout of this graph is governed by `relative` and `parallel` constraints. One `parallel` constraint (purple box node) constrains three sphere object (blue sphere nodes), and another `parallel` constraint constrains six sphere objects. The former arranges three objects at the bottom of the tree in Figure 6.5, and the latter arranges six nodes at the middle of the tree in Figure 6.5. We can easily recognize such structure by looking the expanded constraint graphs. In addition, clicking these constraint nodes highlights the constrained graphical objects in the picture, which will help the users to find the correspondence between the picture and the constraint graph.

By selecting constraints to be stretched, we can explore the structure of the constraint system. Figure 6.10 shows the graph expanded by setting aside a `relative` and two `parallel` constraints from the constraint graph. In this graph, we can see two `average` constraints (yellow boxes) are connected to `box` object nodes (blue spheres). The left yellow `average` constraint has four edges which constrains the highlighted objects of the lower sub-tree in Figure 6.5. Another `average` constraint node has seven connections which means that one object should be put at the average position of other six objects. It constrains the objects at the upper part of the tree in

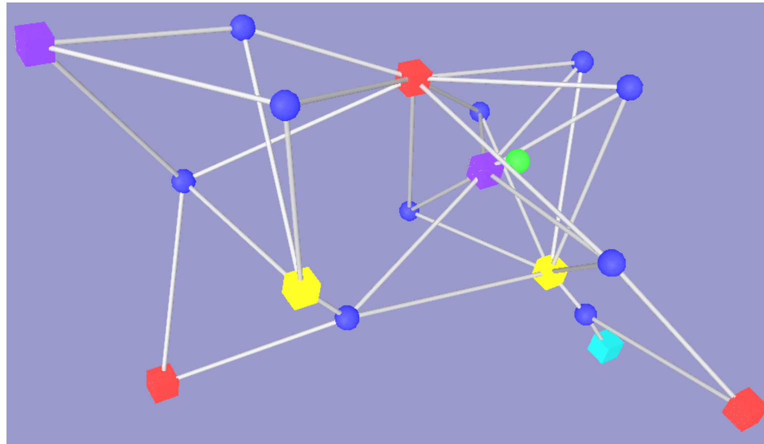


Figure 6.7: A constraint graph of a tree in Figure 6.5 (without line constraints).

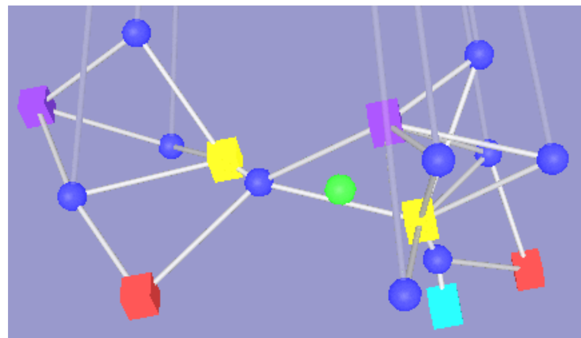


Figure 6.8: A constraint graph — a `relative` constraint is set aside.

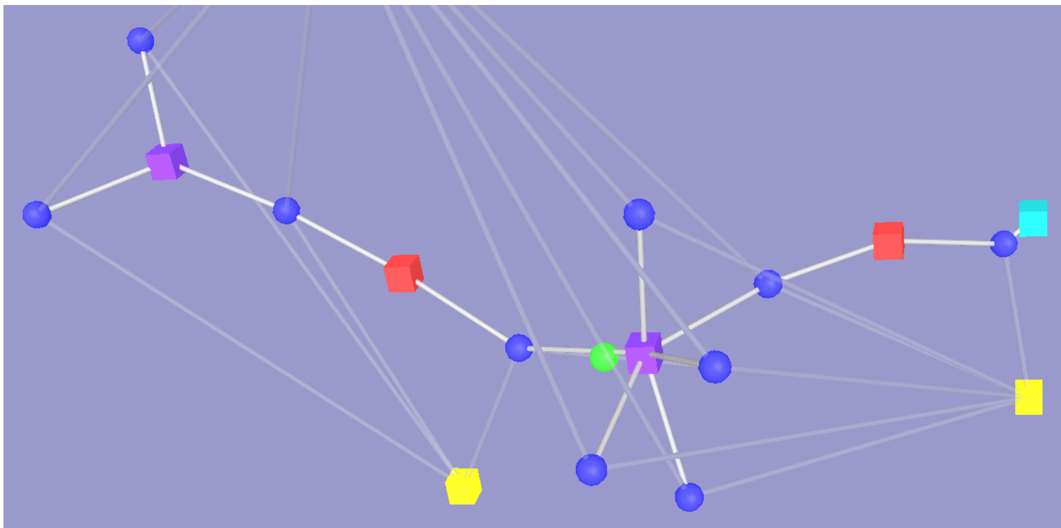


Figure 6.9: A constraint graph — a `relative` and two `average.` constraints are set aside

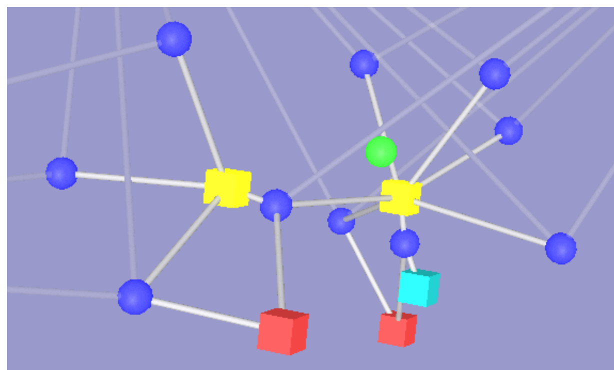


Figure 6.10: A constraint graph — a `relative` and two `parallel` constraints are set aside.

Figure 6.5, that is, the root node should be put at the average position (in x-coordinate) of its six children.

Another method of exploring 3D constraint graphs is to bind up the constraint nodes that constrain same set of graphical objects. As shown in Figure 6.6, there are cases that graphical object nodes are connected by the same set of constraint. Such a set of constraints can be thought of as a compound constraint. The tool provides a command to change the layout of the graph so that the grouped constraints are positioned at almost same place. They are shown to the user as if it is one constraint. The command can be executed on the groups of constraints nodes the user selected, or all groups of constraints. Figure 6.11 shows a 145-node constraint graph of the figure that represents the data structure (two quad-trees) of N-Body simulation program. Figure 6.13 shows a target picture. Figure 6.12 shows the graph where all groups of constraints are bound up. In this constraint graph, object nodes are basically connected by a set of three `relative` constraints. In Figure 6.12, each set of three constraints looks like one constraint, and we can see clearly that the the structure of the constraint graph also contains quad-trees. This is a way of abstracting constraint graphs. We are planning to provide more ways to abstract the graphs, such as to classify and color the groups of constraints, or to abstract the hierarchical structures of constraint graphs.

6.2.3 Implementation of 3D Constraint Graph Visualizer

Figure 6.14 depicts an example process of visualization in our system. The source data are four terms: three `nodes` and a `tree`. They are mapped to four spheres and four graphical constraints by a set of visul mapping rules[3]. The VSR data are then translated into pictures in two ways. First, a tree picture is normally generated by solving the constraints in the VSR data (the left tree in Figure 6.14). Second, the system generates another set of VSR data that represents the structure of the constraints in the original VSR data. The graphical objects in the original VSR data are converted to vertices (sphere nodes) in the constraint graph. The graphical relations are also converted to vertices (box nodes). Edges are generated so that they connect each graphical relation (constraint) and graphical objects constrained by it. The picture of a constraint graph is then generated from the VSR data (the right graph in Figure 6.14). According to the bi-directional translation model, this visualization of the constraint system can be thought of as another way of translating VSR into PR. Usually, VSR data are translated into the picture that they directly represent. On the other hand, this 3D graph visualization translates VSR data into a picture that represents their structure.

[3]For brevity, we assume two-dimensional layout.

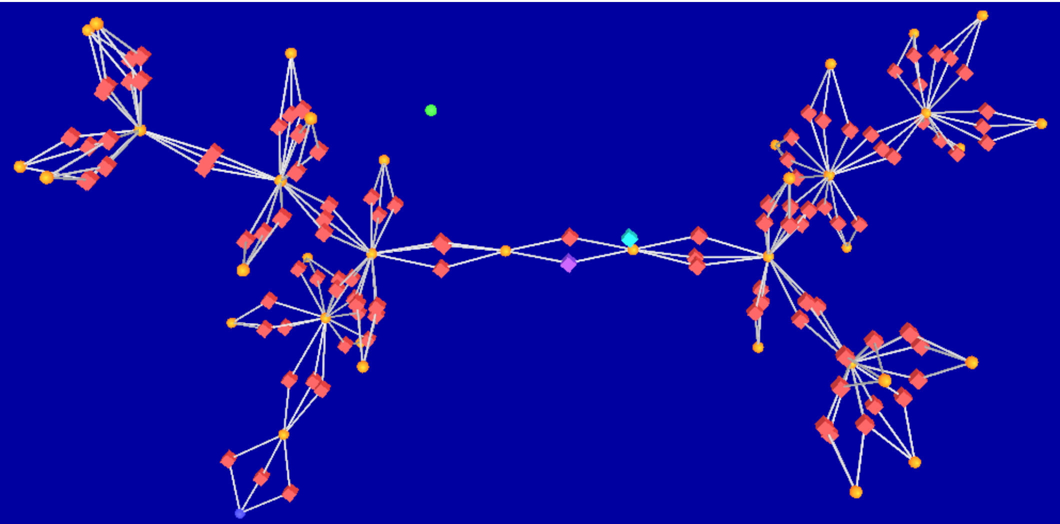


Figure 6.11: A constraint graph of N-body animation.

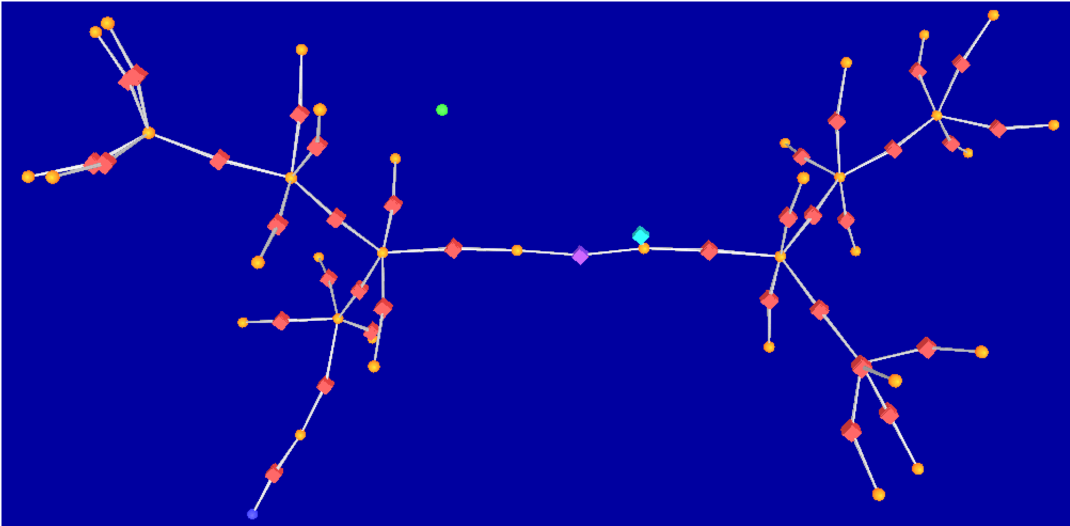


Figure 6.12: A constraint graph — Edges are bound up.

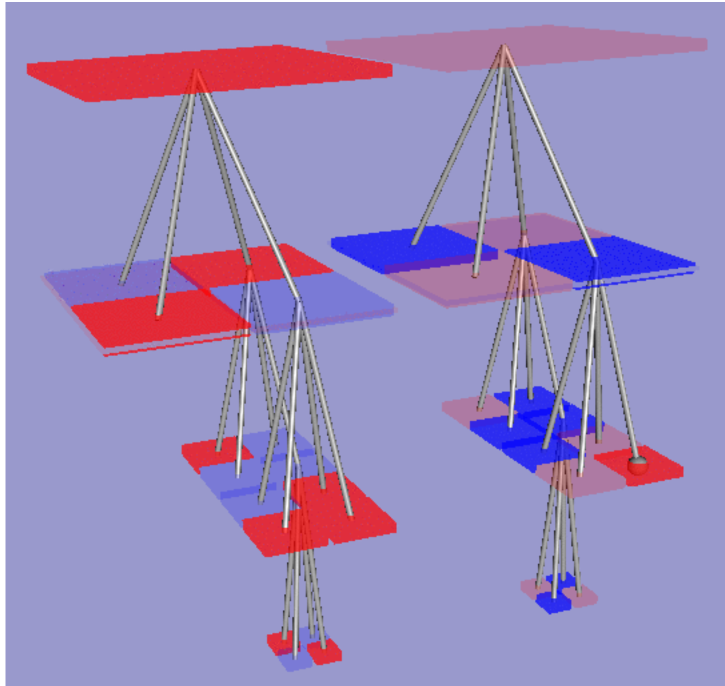


Figure 6.13: Target picture — Two quad-trees.

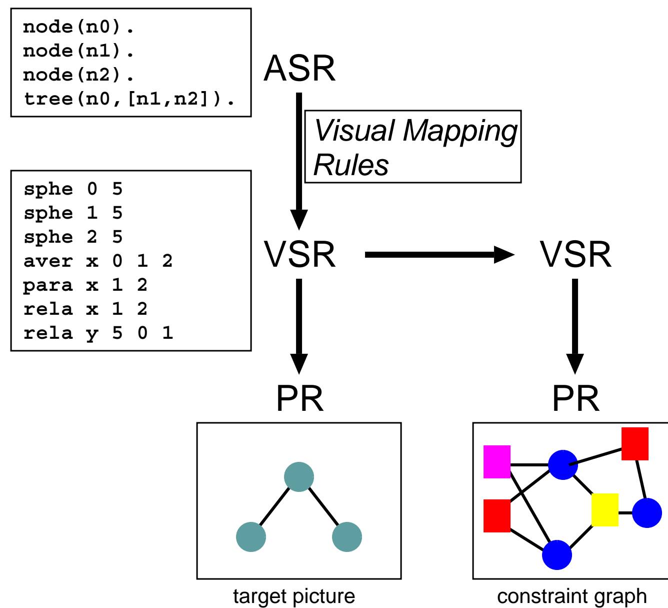


Figure 6.14: An example of translating ASR into PR via VSR.

The graph layout module of our system uses the three-dimensional version of Kamada's graph-drawing algorithm[66], which tries to make the geometric distance between each pair of vertices in the graph close to the logical graph-theoretic distance between them. By utilizing the features of this algorithm, the change of layout described in this section can be easily achieved:

- By setting the default length of all edges from a node very long causes the difference of graph-theoretic distance unimportant, which makes the effect of the node to the layout very little.
- Binding up groups of constraints can be achieved by setting very short edges among a group of constraint nodes.

The graph layout module can lay out graphs three-dimensionally or two-dimensionally. In the current implementation, two-dimensional layout of the graph is achieved by initially placing each node on the x-y plane (z = 0).

6.3 Animating Freedoms in a Constraint System

6.3.1 Overview

Visualization of constraint graphs shows their structure directly. However, it is still difficult to understand the role of each constraint in the whole graph, because constraints and objects are represented abstractly. For example, a lack of constraints may be evident in the visualized constraint graph, but how it affects the result is difficult to guess. To represent the behavior of constraints more directly, we propose another method of visualizing constraint systems. This method animates the target picture itself, and shows degrees of freedom in a constraint system.

For example, if the x and y positions of a box are determined but the z position is not constrained, the box can move along the z-axis while satisfying all constraints. In this case, the system shows an animation that moves along the z-axis and comes back to the original position. The user knows immediately from the animation that the z position of the object is not constrained.

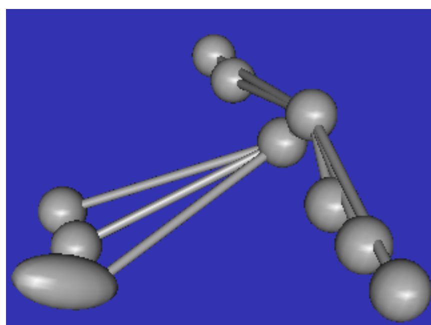


Figure 6.15: Pulling a node in a tree — Movable in this direction.

Figure 6.15 is a screenshot of the animation when the system pulls the node at the bottom-left position to the left. The node is stretched and moved to the left, which means that the node has a degree of freedom in this direction. The two nodes at the bottom also move similarly to the left, which means that the two nodes and the pulled node are constrained to be in a line. The lines that connect them and their parent are also animated, because there are "connect" constraints that connect these nodes.

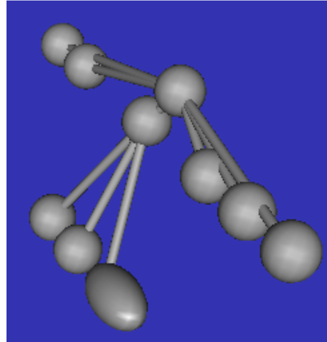


Figure 6.16: Pulling a node in a tree — Immovable in this direction.

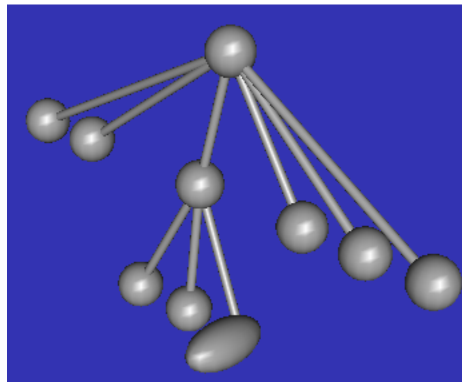


Figure 6.17: Pulling a node in a tree — All nodes are well-constrained.

On the other hand, in Figure 6.16, the system tries to move the same node in another direction, but the node is only stretched and does not move. This indicates that the node is constrained and does not have freedom to move in that direction.

The VSR data that generated the trees shown in Figures 6.15 and 6.16 were made by intentionally removing two constraints from the original VSR data corresponding to the tree in Figure 6.17. The nodes in Figure 6.17 do not have any degrees of freedom, so the node is only stretched and does not move in any direction.

As shown in these figures, even when the object does not move, the object is stretched in the direction in which it is pulled. This shows effectively that the system is trying to pull the object, but that the object cannot move in this direction. If the system showed only the animation of objects that have a degree of freedom, the user might feel that all objects have a degree of freedom to move.

6.3.2 Implementation of the Freedom Viewer

We have implemented the above method into our VSR viewer. When the user clicks on the button named *start animation*, the viewer starts the animation that depicts the degrees of freedom (DOF) in the constraints of the VSR data.

Visualizing DOF is achieved in two steps: (1) detecting DOF in a constraint system; and (2) showing the detected DOF.

Detecting DOF in a Constraint System Detecting degrees of freedom (DOF) in a constraint system means searching for the less constrained variables in a constraint system. Our system utilizes the HiRise constraint solver to search for such variables in the following way.

First, the system randomly or successively selects a variable (attribute) of a graphical object in a set of VSR data. Currently, the system selects only the variables that represent the x-, y-, and z- coordinates of objects. Then, the system checks whether the selected variable has a degree of freedom, as follows:

1. The constraint solver solves the constraints in the VSR data, and assigns a value to each variable according to the solution. At this point, all variables have a value.

2. The system changes the current value of the selected variable, i.e., the system adds or subtracts some constant value to/from the selected variable.

3. The system adds a new *stay* constraint on the selected variable to the constraint system. A stay constraint is a constraint that constrains a variable to hold the current value. The stay constraint is made *weaker* than the other constraints. Therefore, the added stay constraint weakly tries to constrain the changed value of the selected variable.

4. The system solves the modified constraint system again.

5. If the value of the selected variable goes back to the original value, it means that the added weaker constraint is not satisfied, i.e., it conflicts with other stronger constraints. Therefore, the variable is constrained by some constraints in the original constraint system; i.e., the variable does not have any degrees of freedom.

   On the other hand, if the value of the selected variable is not changed after the re-solving of constraints, this indicates that the weak stay constraint does not conflict with other stronger constraints. Thus, the variable is not constrained by any other constraints, and it has a degree of freedom.

Our system does not yet handle inequality constraints. Therefore, a variable either does or does not have a degree of freedom. In future work, we will handle inequality constraints, in which case a range of valid values for each variable must be determined.

Showing a Degree of Freedom According to the checked DOF of each variable in VSR, the system shows an animation to indicate the DOF of each variable. The animation is achieved by moving the graphical objects in the picture visualized from the target VSR data. How an object is moved differs according to whether the variable has a degree of freedom.

- When a variable has a degree of freedom, the system shows an animation generated by changing its value. More precisely, the system generates an animation by gradually changing the value of the target variable to a slightly changed value, and then gradually changing it back to the former value.

  For example, Figure 6.15 shows a screenshot of an animation that shows pull-and-release of a node in a tree. During this animation, the system is solving the entire constraint system repeatedly. This is done by adding an *edit* constraint[51] that is stronger than the other constraints. Using an edit constraint, HiRise can efficiently solve the constraints repeatedly with the value of the target variable changing gradually.

- Even if a variable does not have a degree of freedom, the system shows an animation indicating this. In Figure 6.16, the object is stretched in the direction of the pull, but the position of the object does not change. This animation implies that the system tries to pull the object, but the object does not move because it is constrained. This is a kind of cartoon technique used to distort characters in cartoon animations[76].

Besides animating DOF in a constraint system, the system allows the user to drag graphical objects directly with a mouse. The dragging of a graphical object is executed with satisfying constraints on it. The well-constrained objects cannot be dragged. The system repeatedly solves the constraint system during the user's dragging.

The HiRise Constraint Solver This system uses the HiRise constraint solver[51]. The original TRIP uses an ordinary linear equation solver, and it cannot handle over/under-constrained systems. This is a serious problem, because the system cannot generate a picture from constraint systems that include bugs. HiRise can solve hierarchical linear constraint systems quickly. HiRise tries to solve over-constrained systems by satisfying as many stronger constraints as possible. In addition, the initial values of each variable work as the weakest *stay* constraints, so HiRise can handle under-constrained systems. Therefore, using HiRise, the TRIP system can generate a picture even when the visualization rule has bugs.

Our system does not use hierarchy of constraints except for animations. To animate objects to depict their degrees of freedom, the system introduces a stronger edit constraint to change the value of a variable that has a degree of freedom.

6.4 Related Work

In Thomas's work [18], the distortion effect is used when dragging objects in drawing editors, which makes users feel as if they are dragging "soft" objects. For example, if the vertex of an object is pinned at a point, the user cannot drag it freely but can pull the object to stretch or squash it.

After releasing the mouse button, the object returns to its normal shape. Without such an effect, users cannot easily determine whether an object is constrained and therefore unable to move, or the system is not responding to the user's operation. Our system uses similar techniques, but is extended to handle constraint systems. In addition, our system animates a constraint system without user operations.

6.5 Concluding Remarks

We have described two approaches to visualize the visualization rules of TRIP systems. One is to draw a three-dimensional graph structure of the constraint system in the rules, and the other is to animate the target picture to show the degrees of freedom of graphical objects. We have prototyped these two approaches and applied them to the tree example. Although we have not yet evaluated these approaches, they both help to clarify the structure and behavior of constraint systems.

Chapter 7

Conclusions

We have described a framework for developing kinds of graphical user interface software, designated as a bi-directional translation model. This framework models the general process of visualization, picture interpretation, and animation generation whose domain is mainly abstract relational structures such as trees and graphs. The framework supports the development of systems for building (1) interfaces for direct manipulation of diagrams that represents abstract application data, and (2) animations that shows the changing abstract data in the executed applications. Based on the framework, we have built three systems — TRIP2, TRIP2a, and TRIP2a/3D, which are described in Chapter 4 and Chapter 5

In summary, the contributions of this thesis are as follows:

An Integrated Framework (Chapter 3) We have proposed an integrated framework of visualization, direct manipulation, and animation of abstract application data. It models the general process of these functions. The key idea is that these functions are *translations* between different data representations. The model introduced four data representations: AR (Application Data Representation), ASR (Abstract Structure Representation), VSR (Visual Structure Representation), and PR (Pictorial Representation). ASR is the representation in our model that represents the structure of application data, and it does not have explicit visual appearance information. On the other hand, the purpose of VSR is to represent the high-level structures of pictures. Visualization is a translation from AR to PR via ASR and VSR. Interpretation of figures is an inverse translation from PR to AR via VSR and ASR. The mapping between ASR and VSR (*visual mapping*) is the translations that determine how to visualize abstract data and how to interpret figures. Only by changing declarative visual mapping rule sets that specify visual mapping between ASR and VSR, the programmer can try various pictorial representations of abstract application data. The programmer must devise visual mapping rules appropriately so that users can easily understand the generated pictures. As users understand the meaning of abstract application data via pictures, visualized pictures should *represent* abstract application data, and thus they should have a similar structure. As both structures are similar, visual mapping rules are usually simple.

We have also integrated animations into our framework by naturally extending it to the time dimension. In the framework, animations are regarded as *operations* on PR that are translated from operations on AR via operations on ASR and VSR. The translation among operations is executed maintaining consistency with the mapping relations among the AR, ASR, VSR, and PR data. We have chosen an interpolation-based method for implementation of the extended framework for animations. That is, animations are generated by interpolating a sequence of pictures translated from the running application's internal data. Rather than

directly specifying procedural specifications of animations (motions or transformations), the programmer specifies declarative transitional operations, i.e., the methods used to determine how to interpolate a pair of successive pictures in the sequence of pictures. Altogether, the programmer specifies animations by two types of mapping rules: visual mapping rules and transition mapping rules. The programmer has not necessarily specify transitional operations. Default transitional operations are prepared for that purpose.

Tools implemented based on the framework (Chapter 4, Chapter 5) We have described three systems implemented based on the framework. One is the TRIP2 system — a system for developing interfaces that provide direct manipulation of abstract application data. The other systems — TRIP2a and TRIP2a/3D — are for making animations.

TRIP2 is a system that achieved the bi-directional translation between AR and PR. Using TRIP2, the user can edit diagrams visualized from abstract application data to modify the corresponding application data. That is, the application data are visualized to the corresponding diagrams, and the diagrams modified by the user are interpreted by the system and converted back to the application data. The programmer can build such interfaces mainly by specifying visual mapping rules that describe how to visualize abstract application data and how to interpret diagrams modified by the user. TRIP2 supposes the domain of application data is mainly relational structure data such as hierarchical structure data and network structure data. The target diagrams are those that consist of graphical objects like boxes and circles connected by lines, and can be arranged by a linear constraint solver and a undirected graph layout module, that is, diagrams such as trees and graphs. TRIP2 is implemented on NeXT computer using NextStep, Objective-C, and Prolog. The bi-directional mapping modules are implemented with Prolog, and the programmer writes mapping rules in Prolog. It is possible to write an application in Prolog and make its direct manipulation interface with the TRIP2 system. We applied TRIP2 to build several examples. For example, we built interfaces for a graph structure, a kinship diagram, an ER-diagram, and a simple Othello game application.

TRIP2a is our first system based on our framework for constructing abstract animations that depict the behavior of program executions. Animations are generated by interpolating the sequence of pictures generated by translating the sequence of abstract application data collected from the executed applications. The way of visualizing internal data of an application is specified in the same way with TRIP/TRIP2. That is, the programmer writes visual mapping rules to determine how to visualize abstract application data. It is also possible to specify *transition mapping rules* between abstract operations on ASR corresponding to the operations in the executed applications and the transitional operation on VSR that are the methods for interpolating two successive pictures. Transition mapping determines the way of transforming (moving, scaling, ...) graphical objects in an animation. By changing transition mapping rules, the programmer can easily changing the behavior of graphical objects in an animation. TRIP2a is also implemented on NeXT computer. The implementation of visualization module that generates sequences of pictures from ASR data is basically same as TRIP and TRIP2. Therefore, the programmer can use basically same visual mapping rules that are used in TRIP/TRIP2 for making an animation from a sequence of ASR data. We have made various algorithm animations using TRIP2a. For example, we made various sorting algorithm animations, bin-packing algorithm animation, animations of data structures (tree, graph, ...), and a minimum-spanning-tree(MST) algorithm animation. TRIP2a is integrated with TRIP2 so that we can use two functions together: the bi-directional translation between abstract data and pictures, and the translation from a sequence of abstract data into an animation, that is,

direct manipulation of abstract application data and animation of abstract data in the executed application. Using this function, it is easy to draw a diagram as an input to the target program of the algorithm animation. For example, in the example of MST algorithm animation, the user draws a graph as an initial input to the MST-finding program.

TRIP2a/3D is the successor of the TRIP2a system that specializes in generating animations. It is implemented to handle three-dimensional representations and event-driven animations, which is useful for animating parallel program executions. Its implementation is separated to the mapping module and the viewing module, because it is convenient to implement viewers on various platforms, such as on MS-Windows, on X-Window system, and on Java3D. The mapping module is developed with KLIC, and the programmer writes mapping rules in KLIC. We have described several algorithm animations and visualizations of program executions generated with TRIP2a/3D such as N-queen problem animation and the animation that show quad-trees in 3D.

Methods for Browsing Constraints (Chapter 6) Visual mapping rules have great importance in our TRIP systems. It is declarative and usually simple to write. However, because the programmer must combine geometric constraints in the rules to arrange graphical objects, it is sometimes difficult to debug the rules. It is necessary to support the debugging of visual mapping rules with our systems. As a step toward solving this problem, we have described techniques for browsing a constraint system in VSR data to understand the structure of the constraints.

One way is to show constraint systems as two- or three-dimensional graphs. The nodes in the graph are constraints and graphical objects in a constraint system. A constraint is represented as a box, and a graphical object is represented as a sphere. Edges are set up so that they connect constraints to the graphical objects constrained by them. By looking at constraint graphs, the user can see the structure of constraint systems more directly than in their textual form. We used Kamada's spring-model algorithm to calculate the layout of constraint graphs. In addition, we utilized this algorithm to change the layout of the graph and focus some constraints in the graph. It is achieved by lengthening the edges connected to the constraint which should be unfocused. We have shown the example constraint graph of a simple tree picture, and described how its layout is changed to focus/unfocus some constraints in the constraint system.

Another way of showing constraints visually is to animate constrained graphical objects. The system depicts degrees of freedom in the positional constraints in the following way: Basically, the system tries to *pull* each graphical objects along the x-,y-,z-axis. If the object is not constrained by the constraints, the system succeeds to move the object, that is, an animation that the object moves a little is shown to the user. On the other hand, if the object is well-constrained, the object cannot move because of the constraints. In that case, an animation is shown to the user that even if the object is pulled, it cannot move in the pulling direction. This situation is represented using the technique of cartoon animations — distortion. By stretching the shape of the pulled object, the system depicts that the object is pulled but cannot move in that direction. We have created an example and described these animations. These techniques are useful for understanding and debugging constraint systems in visual mapping rules.

In the appendix, we described the TRIP3/IMAGE systems – systems that can interactively make visual mapping rules from visualization examples. In addition, the IMAGE system solved the implementation problem of TRIP2. That is, in the IMAGE system, the same rules can be used both

for visual and inverse visual mapping rules. We also described the details of TRIP2a/3D, and the examples of mapping rules for TRIP2a in the appendix.

Future Work As a future work, we are planning to re-build the development environment based on our framework to improve and integrate the prototype systems described in this paper. The current implementations of TRIP systems have problems, and there are several issues on the design of a new system. The challenges of improving current implementations are as follows:

Improving application interface The current implementation of the TRIP2 and TRIP2a has the following application interfaces:

TRIP2 To utilize TRIP2 to build an application, the programmer must use Prolog to write its program. The database of the Prolog system is shared among TRIP2 and the application so that they can pass data to each other.

TRIP2a There are three ways for applications to pass data to the TRIP2a system.

- Use log files. The application writes out snapshot of application data as textual log data. TRIP2a/3D uses only this interface.
- Use RPC-like mechanism on NextStep. In the NextStep development environment, Speaker/Listener classes are provided for messaging between application processes. The programmer can use the Speaker class to pass application data to TRIP2a. However, the programmer must use Objective-C to write application programs.
- Write application programs in Prolog. Like TRIP2, the programmer can write an application with visual mapping rules. Through the Prolog database shared by TRIP2a and the application, the application can pass its data to TRIP2a.

More elaborate application interface is desirable. This problem is related to the definition of ASR. In TRIP2, ASR is a set of facts in Prolog. In the IMAGE system — the successor of TRIP2, the programmer defines the types of ASR data in Lisp. Therefore, it is difficult to use other popular languages such as Java and C++ to build applications.

Improving the method of specifying declarative visual mapping rules The bi-directional translation between ASR and VSR is the heart of our framework. In TRIP2/TRIP2a, it is implemented in Prolog. In TRIP2a/3D, it is implemented in KLIC. In the IMAGE system, it is implemented in CommonLisp. Thus, the visual mapping rules are written in Prolog, KLIC, and CommonLisp, respectively[1]. Since the descriptive power of languages are different, it is desirable to be able to specify mappings in multiple ways from interactive specification to full-fledged programming. The translator should be changed according to the mapping rule used by the programmer.

Adding more types of geometric constraints to VSR Our systems use various constraint solvers, but basically they have a linear equation solver and a graph layout module. The IMAGE system utilizes the constraint hierarchy mechanism in the process of generating mapping rules.

More types of constraints should be added to VSR. For example, non-linear constraints are useful for specifying parallel and distance constraints. Another useful type of constraints is

[1] In fact, in the IMAGE system, the programmer does not write textual mapping rules directly. The rules are generated by the system from the examples provided by the programmer.

energy-based constraint. We can define the energy of the layout of objects so that the energy is low when the layout is desirable and the energy is high when the layout is not desirable. Using this idea, various constraints can be regarded as energy functions of the layout, and solving these constraints means minimizing the energy. The constraint provided by the undirected graph layout module used in the TRIP systems is an example of such constraints. Such constraints are useful for globally beautifying the layout of pictures.

To cope with these issues, we are considering to implement our framework on Java. That is, every data in the framework is represented as objects in Java. We are to provide the classes as JavaBeans components. Figure 7.1 shows the architecture of our new system. The four data representations in our framework are designed as follows:

AR The application data themselves implemented as Java classes. They may not be intended to be used for visualization and direct manipulation.

ASR We provide a Java interface corresponding to ASR. The programmer defines classes that implement the interface. It is necessary because the mapping between the defined ASR and the VSR is independent to the application data. In addition, it is necessary to collect and represent the application data directly correspond to the target picture. Their histories are stored for making animations, if necessary. The bi-directional translator accesses user-defined ASR model via ASR model adapter.

VSR Graphical objects and graphical relations are provided as Java classes. Graphical relations are solved by the constraint solver, and the result are used to determine the absolute coordinates of graphical objects. A set of graphical objects and graphical relations represent a picture. Their histories are also stored for animations.

We use Chorus[53] constraint solver. By using Chorus, we can provide various types of constraints as graphical relations including linear and non-linear energy-based constraints. It also enables to employ constraint hierarchy mechanism in VSR. It is useful to differentiate important constraints required for the structure of the picture and not-so-important constraints for beautification of the layout. It was also useful for the implementation of the IMAGE system.

PR Graphical objects in VSR are also used as PR. They are shown in the drawing editor provided by the system. Similar to the IMAGE system, the parts of the pictures are inferred from the mapping rules, and provided in the palette window. In addition, it is possible to enable some constraints in the graphical relations when the user edits diagrams. This can be achieved by using constraint hierarchy mechanism in the Chorus constraint solver. It is interesting to make the interaction module exchangeable to powerful popular drawing tools such as TGIF.

As for bi-directional translations, we are to provide several ways to specify visual mapping rules. First, it should be written in declarative languages such as Prolog for the bi-directional translator. Syntax like Constraint Multiset Grammar[82] may be also suitable for this purpose. Second, it should be generated interactively like in the IMAGE system. Third, it may be helpful to be specified by XML stylesheet, if the source data are also represented as XML data.

It is also challenging to design the system on XML infrastructures. XML is a markup language for documents containing structured information. XML is applied widely for web applications. There are research work on visualization of XML data. Hosobe et al. proposed a constraint-based approach to information visualization for XML[54]. Figure 7.2 shows the architecture of their proposed framework. It is closely related to our framework of TRIP systems. The source XML

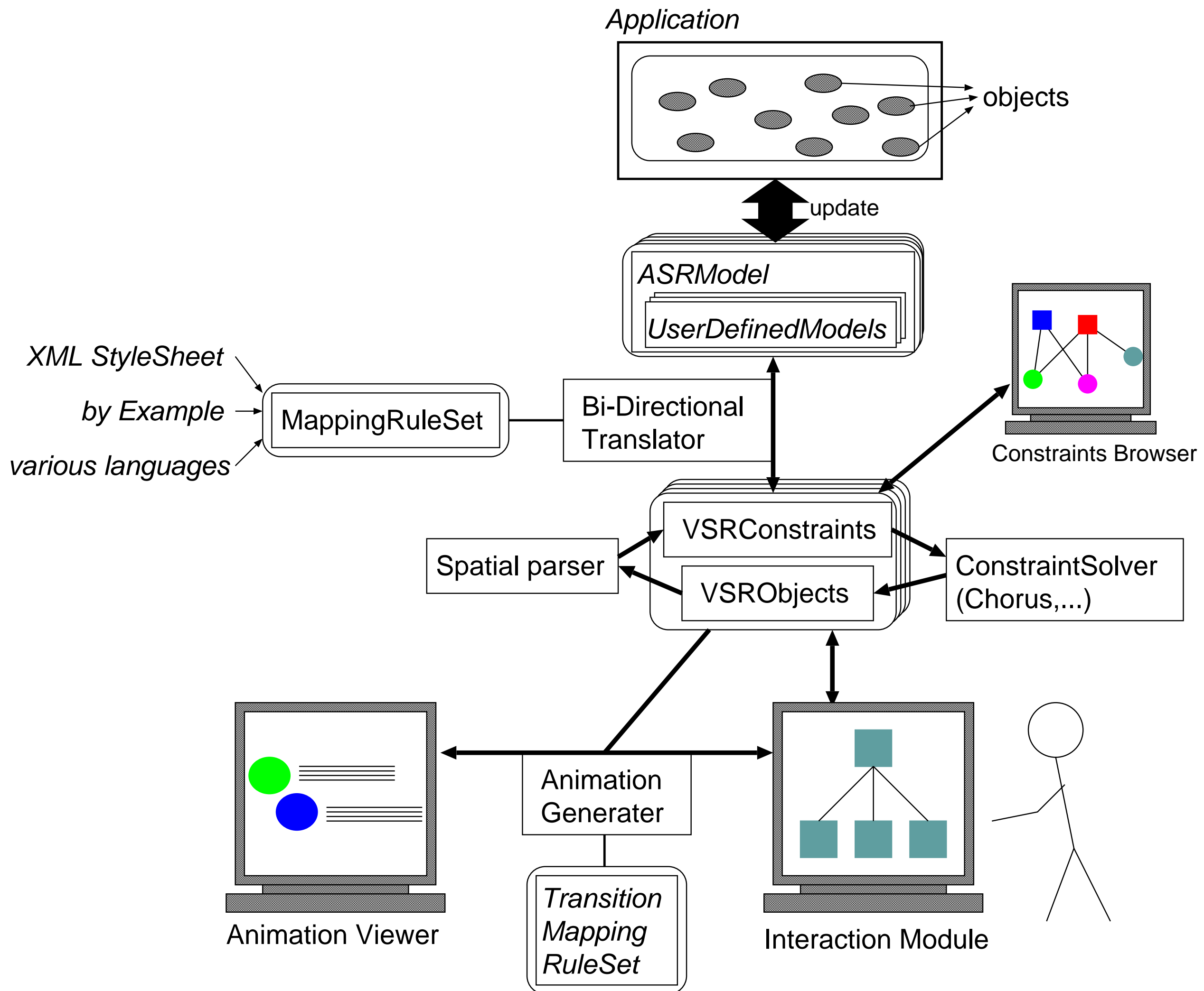


Figure 7.1: The architecture of the next system.

data document corresponds to ASR data in our model. It is translated to a XML-VL[54] document which corresponds to VSR data. The translation is specified by a XML stylesheet for XSLT which corresponds to a set of visual mapping rules. XML-VL documents are translated into pictures by solving constraints in them. ILOG-JViews[106] also uses stylesheet to customize the visualization of application data. It is interesting to extend their work to achieve bi-directional translations. There is a work on inverse transformation of XSLT[97], which may be helpful to achieve the inverse translation from a diagram to the source XML data document.

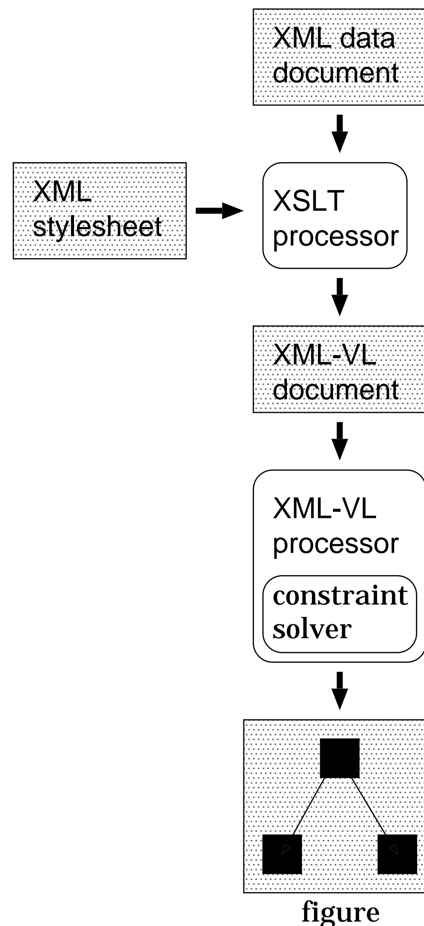


Figure 7.2: Visualization framework with XML-VL[54].

This thesis focused on abstract data and their visual representations, especially figures and diagrams such as trees and graphs. However, the framework described here may also be applicable to other types of data and representations. For example, in *computer vision* (CV), the target visual representation is image data. Image recognition can be regarded as translations from image data. Gesture recognition and motion analysis can also be regarded as translations from video data, i.e., sequences of image data. The same approach as described in this thesis may be applicable to these functions, which may help to implement libraries or tools to support construction of CV applications. *Artificial Reality* (AR)/*Mixed Reality* (MR) may be another example. These systems use a kind of "translation" from various kinds of information to image and video data.

Presentation and recognition of various visual representations are key functions to realizing ideal visual and interactive interfaces between users and computers. The principles presented in this thesis will help in the construction of true interactive and visual software, which will enable better

visual and interactive communication between users and computers.

Appendix A

Generating Visual Mapping Rules by Examples

Chapter 3 introduced the bi-directional translation model, which is a common architecture for interfaces that achieves direct manipulation of abstract data algorithm animation. According to the model, we created TRIP2 and TRIP2a, which are tools to support the development of interface software. Using these tools, programmers can develop interfaces by writing only a set of visual mapping rules. Nevertheless, it is still not easy for non-programmers to write visual mapping rules properly, because they must be written in textual rules with no intuitive link to the target visual representation.

To cope with this problem, we proposed a Programming by Visual Example (PBVE) scheme, and developed two novel systems. The first was the TRIP3 system [88], which generates a visual mapping rule set from a pair of ASR data and picture data. The successor of TRIP3 was the IMAGE system [87], which can exploit multiple pairs of examples and supports the input of examples by the programmer.

Although these projects were not the primary subjects of this thesis, they are reviewed here for reference because they are projects performed within our group and have led to proposed solutions to some of the issues with the TRIP2/TRIP2a systems described in this thesis. We describe these systems briefly by citing examples of the generation of visual mapping rules from examples.

A.1 TRIP3

A.1.1 Overview

TRIP3 is an environment for generating visual mapping rules by providing an instance of intended mapping, i.e., a pair of ASR data and their visual representation. This system is based on the framework called Programming by Visual Example (PBVE), which is a framework for generating visual mapping rules from a visualization sample.

Figure A.1 illustrates an example of generating a visual mapping rule with PBVE. The process of generating rules consists of four steps:

1. Programmer's Input of Mapping Instance
   The programmer inputs a pair of ASR data and the corresponding picture.

2. Extracting VSR
   The VSR data are extracted from the drawn picture.

3. Object Generalization
   The objects in the ASR data and the VSR data are generalized.

4. Rule Generation
   The system generates mapping rules using templates.

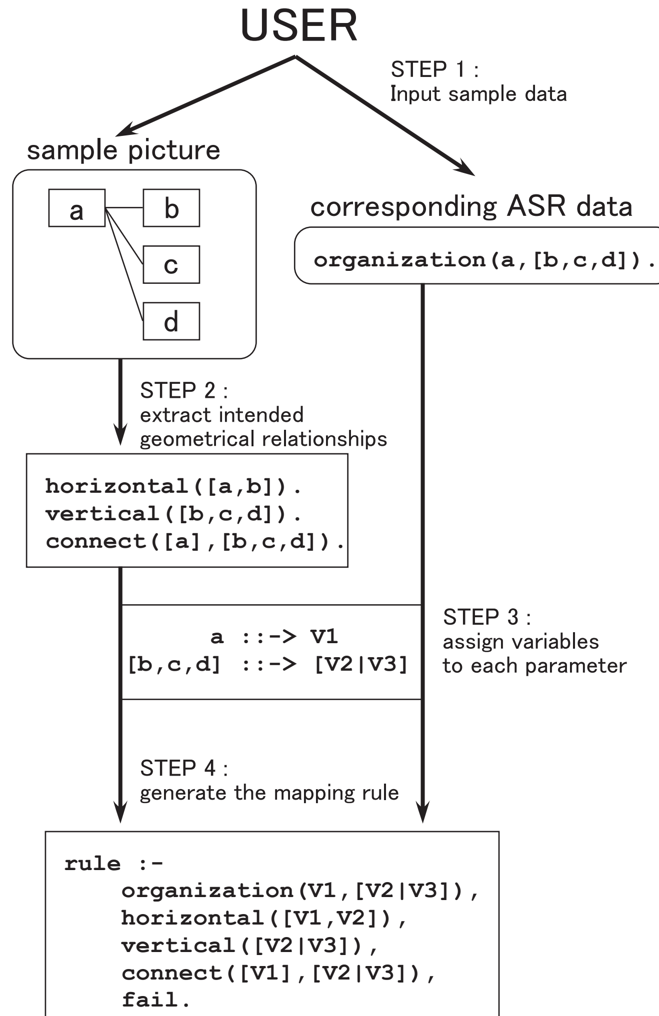


Figure A.1: Programming by visual example — a process of generating visual mapping rules.

We describe these steps in the following paragraphs.

Programmer's Input of Example Data The first step of PBVE is the input of example data by the programmer. The programmer provides the system with a pair of examples; the example ASR data and its corresponding picture that is an instance of the mapping intended by the programmer. In Figure A.1, a tree picture and the term `organization(a,[b,c,d])` are input by the programmer.

Here, the programmer intends that there is an organization whose members are `a`, `b`, `c`, and `d` where `a` is the boss, and the others are staff. The tree picture represents this organization. Each rectangle with a label in the tree represents a member of the organization.

Extracting VSR The second step is extraction of VSR data. The system extracts graphical objects and geometrical relations from the given picture. In TRIP3, the programmer draws a picture in the drawing editor, which is integrated with an incremental spatial parser. Every time the programmer draws a new object in the editor, TRIP3 parses the picture and infers geometrical relationships among the drawn objects. The graphical objects are also extracted from the picture at the same time. The process is simple because the programmer selects a type of object before drawing, and TRIP3 can easily infer the types of objects drawn in the editor.

For example, when the programmer draws a rectangle horizontal to the circle already drawn as in Figure A.2, the spatial parser infers `horizontal` relations between them. TRIP3 shows inferred relations to the programmer in two ways. One is to show the relations in the TRIP3 drawing editor. In Figure A.2, the inferred relations are represented as horizontal dotted lines. The other way of presenting inferred relations is to list them in the confirmation window (Figure A.3). The programmer can select relations from the list by checking the corresponding boxes. See reference [88] for more details.

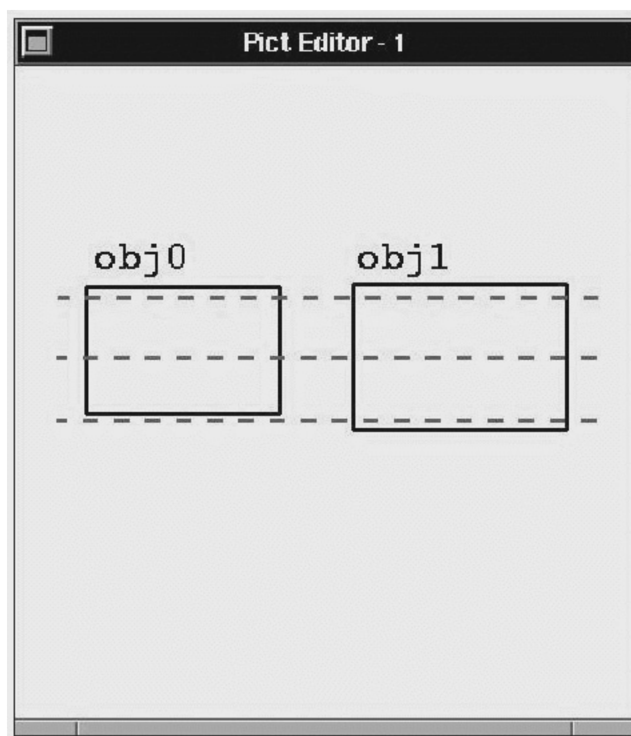


Figure A.2: TRIP3 — picture editor.

Generalization The third step is generalization of the ASR and VSR data extracted at the second step. They represent a typical instance of the target mapping relation, and should be generalized to make a visual mapping rule that represents more general cases.

For example, the ASR and VSR data extracted from the example illustrated in Figure A.1 are shown in the upper half of Figure A.4[1]. Each term represents a relation among `a`, `b`, `c`, and `d`, which are instance abstract/graphical objects. To generate a visual mapping rule, they must be generalized to represent various mapping relations between ASR and VSR data.

[1]In fact, more VSR data are necessary for layout.

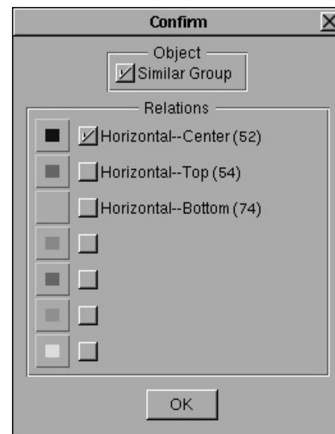


Figure A.3: TRIP3 — confirmation panel.

```
Extracted Data
              ASR data                          VSR data
     organization(a,[b,c,d])  <--> horizontal([a,b])
                                   vertical([b,c,d])
                                   connect([a],[b,c,d])
  -----------------------------------------------------
Generalized Mapping
              ASR data                          VSR data
     organization(V1,[V2|V3]) <--> horizontal([V1,V2])
                                   vertical([V2|V3])
                                   connect([V1],[V2|V3])
```

Figure A.4: An instance of mapping relation extracted from an example (above) and the generalized mapping relation (below).

To achieve this, TRIP3 converts the given ASR and VSR data in two ways:

- Substituting Variables into Atoms

  Each atom in the arguments of terms is substituted with a different variable. An atom in ASR terms represents an abstract "object." An atom in VSR terms is used as an ID of a graphical object. In both cases, atoms in the examples represent specific objects. Therefore, they are converted to variables, which represent arbitrary objects.

- Generalizing Lists

  Lists in the extracted ASR and VSR data must also be generalized. However, simply substituting a variable into each element in the list is not enough to generalize its length. To handle lists of arbitrary length, a ground list term is substituted with a list variable.

In the above case, `a` is converted to `V1`, and the list `[b,c,d]` is converted to `[V2|V3]`. The mapping generalized in this way is shown in the lower half of Figure A.4.

Rule Generation Finally, mapping rules are generated using templates. Figure A.5 shows the template of mapping rules for non-recursive data structures. The body of the template consists of the ASR and VSR terms. The ASR terms generalized at the third step are put into the upper half

```
mapping_rule :-
    asr_term1(X1, X2, ...),
    asr_term2(Y1, Y2, ...),
        ...
    vsr_term1(Z1, Z2, ...),
    vsr_term2(W1, W2, ...),
        ...
    fail.
```

Figure A.5: A template for generating mapping rules.

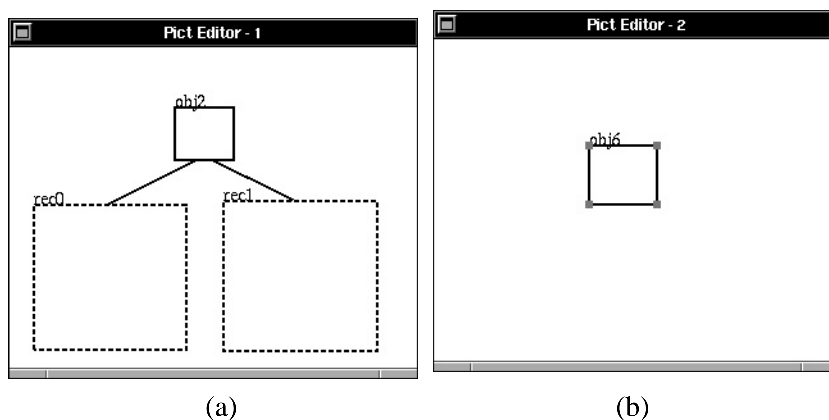
of the template, and the generalized VSR terms are put into the lower half. The generated rule is a Prolog clause, which is used for mapping between ASR data and VSR data. Figure A.1 shows the resulting mapping rule.

A.1.2 Rule Generation for Recursive Data Structures

The method of generating mapping rules for recursive data structures is different from that for non-recursive data structures. To handle recursive data structures, the system must determine which term in the example is the recursive term. In addition, it is difficult to infer mapping rules for a recursive data structure from only one example. To make mapping rules for a recursive data structure, at least two examples are needed, one for a terminal case and another for a non-terminal case. In TRIP3, the programmer specifies the recursive part explicitly using a special primitive graphical object for drawing recursive parts of a picture. The programmer provides two examples, one for a terminal case and one for a non-terminal case.

For example, when creating a set of mapping rules to map tree structure data to a tree, the programmer draws example pictures as shown in Figure A.6. Figure A.6(a) shows an example picture for the non-terminal case. The programmer draws a small rectangle as the root of the tree and two larger dotted rectangles as its children. Dotted rectangles are special objects that represent the recursive parts of the picture.

TRIP3 assumes that each recursive part in a picture has a counterpart in the example ASR data. The programmer uses the name of the rectangle (`rec0` and `rec1`) to represent the recursive part of



(a) (b)

Figure A.6: TRIP3 — an example of tree drawing.

the ASR terms (Figure A.7).

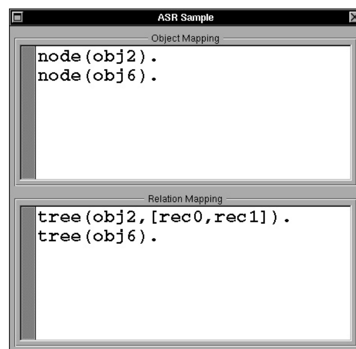


Figure A.7: TRIP3 — ASR example.

In addition, the programmer also has to provide an example for the terminal case. Figure A.6(b) shows the picture, a rectangle named `obj6`, for the terminal case. The programmer provides corresponding ASR data using the editor (Figure A.7). From these given examples, TRIP3 generates a set of mapping rules for tree diagrams, which is shown in Figure A.8.

```
%% Rule for the recursive case
visualize(tree(X, [H|L])) :-
   recursive([H|L]),
   box(X),
   horizontal([H|L]),
   x-center(X,[H|L]),
   y-order([X,H]),
   fail.
%% Rule for the terminal case
visualize(tree(X)) :-
   box(X),
   fail.
```

Figure A.8: TRIP3 — recursive mapping rule.

A.2 IMAGE

A.2.1 Overview

The IMAGE system is the successor of TRIP3[87], and is based on the framework called *Programming by Interactive Correction of Example*. The problems of TRIP3 on which IMAGE has focused, which are also problems of other PBE systems, are as follows:

- The mapping rules generated by the TRIP3 system are represented in system-specific textual forms, which make it difficult for programmers to understand the rules afterwards. Thus, it is difficult to check whether the generated rules conform to the programmer's intentions.

- There is no way to revise generated rules interactively. The generated rules are represented as text and the programmer has to revise textual mapping rules.

- Programmers often have difficulty deciding what examples to provide to PBE systems to effectively generate the rules intended.

  TRIP3 infers mapping rules from only one example of ASR and VSR data. Thus, it is particularly difficult to determine what example should be provided.

IMAGE tries to solve these problems in the following ways:

- To exhibit generated mapping rules to programmers, the system shows the picture visualized with them rather than showing their textual form directly to programmers. The programmer then decides from the picture whether the generated rules are satisfactory.

- The programmer corrects a mapping rule by modifying the picture presented by the system. The programmer repeatedly corrects the picture until the system begins to generate appropriate pictures from various ASR data. When the programmer corrects the picture, the system revises the mapping rule so that it generates the correct pictures.

- The system automatically produces a series of example ASR data and displays the corresponding example pictures to the programmer. The programmer is only asked whether the presented pictures are correct, and adjusts them if they are not satisfactory. Thus, the programmer does not need to decide what examples to provide to the system.

IMAGE is implemented on NeXT using Objective- C, Common Lisp, and the DETAIL constraint solver[55]. See reference[87] for more details.

A.2.2 Interactive Rule Generation Example

In this section, we describe the interaction between the IMAGE system and a programmer interactively generating rules for the organization diagram shown in Figure A.9. We also describe how the problems of TRIP3 are solved by the approach taken by IMAGE. The characters in parentheses at the head of each step correspond to the stages in the process of generating mapping rules illustrated in Figure A.9.

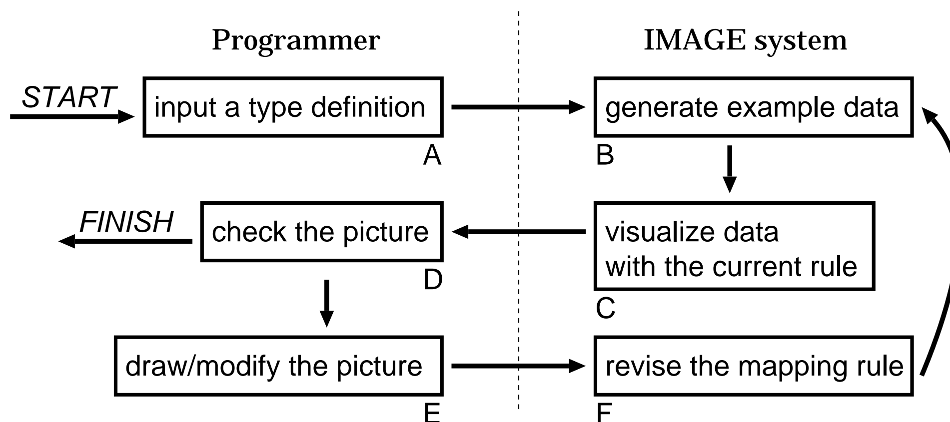


Figure A.9: Interaction between a programmer and IMAGE.

1. **[A]**: First, the programmer inputs the type definitions of ASR as follows:

```
data-type(man){
   name : word;
}
data-type(organization){
   boss : man;
   staff : list-of man;
}
```

Here, two type definitions, man and organization, are entered. The type man has one attribute, name, the type of which is word. The type organization has two attributes; boss and staff. The type of boss is man as defined above. The type of staff is list-of man; that is, staff is an ordered collection of data of type man.

2. **[B]**: According to the type definitions, the system generates the simplest application data example for each data type. For example, the simplest data for the type man is:

   ```
   application-data(man, #1){
      name = "word1";
   }
   ```

   Data are given an identifier; here the identifier is #1. The value of each attribute is automatically determined by the system. Here, the value of name is "word1."

3. **[C→E]**: The programmer draws a visual representation corresponding to the application data presented by the system. As shown in Figure A.10, the programmer drew a rectangle containing the string "word1" in the drawing editor. This is given to the system as a visual representation corresponding to the ASR data presented to the programmer.

   Note that the string "word1" was prepared by the system and placed automatically in the drawing editor before the programmer began to draw a picture. The system guesses and prepares necessary graphical parts to draw a picture from the type definitions entered by the programmer. The programmer uses these to draw a picture. In the above case, as the type man has a string as an attribute, the system infers that a string object is necessary to draw a picture.

4. **[F]**: The system infers a visual mapping rule for the data of type man from the drawn picture and the corresponding application data.

5. **[B,C]**: The system visualizes a slightly more complex application data example using the inferred mapping rule, and presents it to the programmer.

6. **[D]**: The programmer checks the presented visualization. In this case, the programmer is satisfied with the visualization, so the rule generation for the type man is finished. As a result, the system generates a mapping rule that maps between a rectangle with a string and a term of type man.

7. **[A→E]**: Then, the system starts to generate a mapping rule for the type organization. First, the programmer draws a picture corresponding to the example data of type organization presented by the system. The following is the simplest data for the type organization:

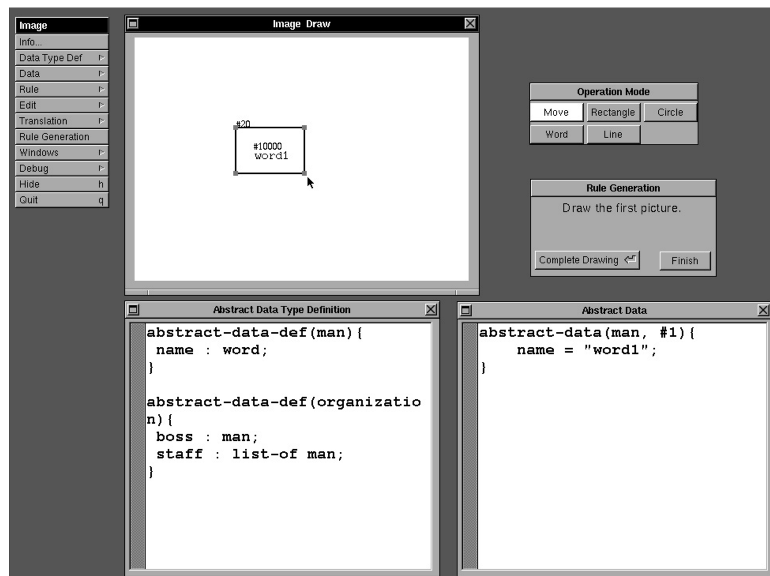


Figure A.10: Screenshot of the IMAGE system.

```
application-data(organization, #10){
    boss = #1;
    staff = [#2, #3];
}
```

The value of `boss` is `#1`, which is the identifier of type `man`, and the value of `staff` is `[#2, #3]` — a list with length two.

As the mapping rule for `man` has already been generated, and the example dataset of type `organization` contains three identifiers of type `man`, the system prepares three `man`s for drawing by the programmer. That is, three rectangles with a string are put in the drawing editor (Figure A.11(1)).

The programmer uses these three rectangles to draw an initial visual representation of the application example data (Figure A.11(2)).

8. **[F,B,C]**: The system generates the initial version of a mapping rule for the type `organization`, uses it to visualize slightly more complex application example data, and presents the resulting picture to the programmer (Figure A.11(3)).

9. **[D,E]**: The programmer checks the presented picture. As it does not satisfy the programmer's intention, the presented picture is modified. In Figure A.11(4), the programmer corrects the picture so that (1) the boss and the staff at the top are aligned horizontally, and (2) a line must be drawn between the boss and each staff member.

10. **[F,B,C]**: The system re-generates a visual mapping rule based on the application example data and the modified corresponding picture. Then, it displays the visual representation of more complex example data visualized by the improved mapping rule (Figure A.11(5)).

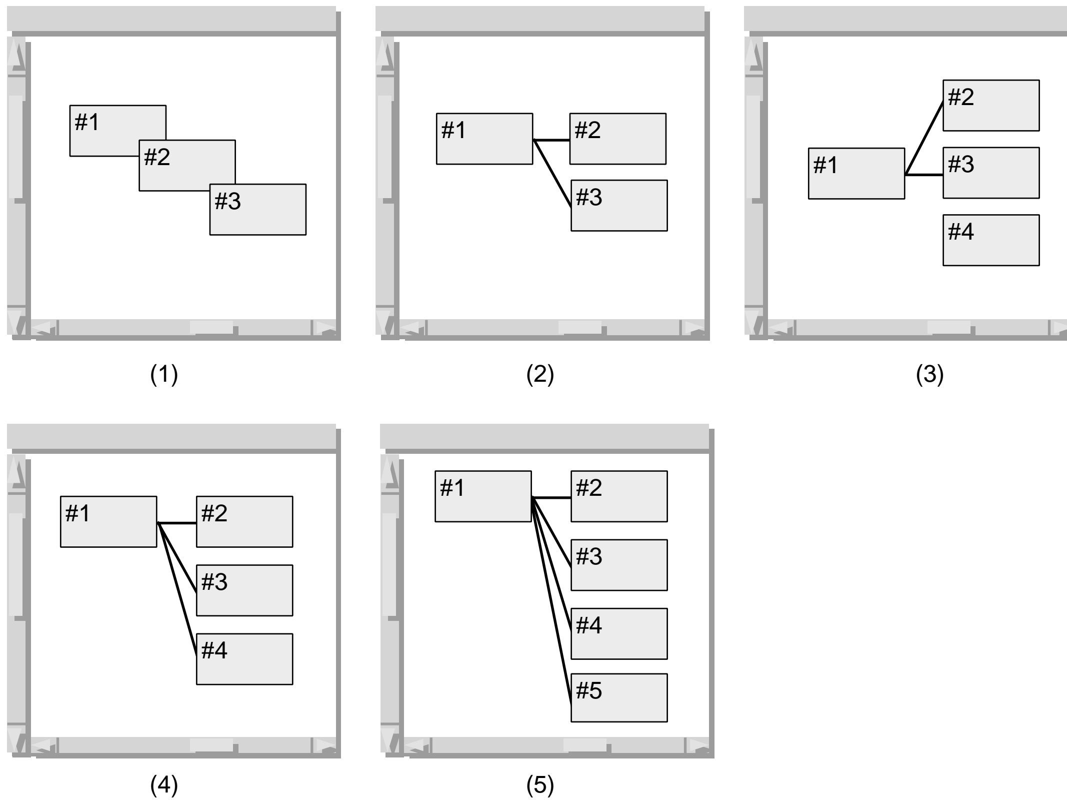


Figure A.11: IMAGE — screenshots of drawing editor in rule generation for “organization”.

11. **[D]**: The programmer checks the presented picture. This time, the programmer is satisfied with the presented picture, and therefore the generation of the mapping rule for the data of type `organization` is finished.

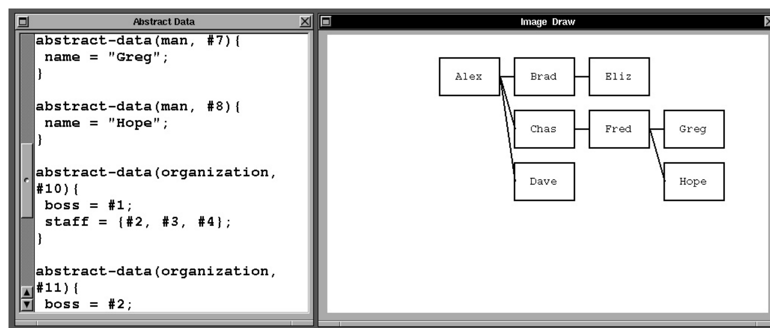


Figure A.12: IMAGE — organization diagram screenshot.

A.2.3 Miscellaneous Issues

Type Definitions Until TRIP3, explicit definitions of ASR were not required. The ASR of an application was defined implicitly by its visual mapping rules.

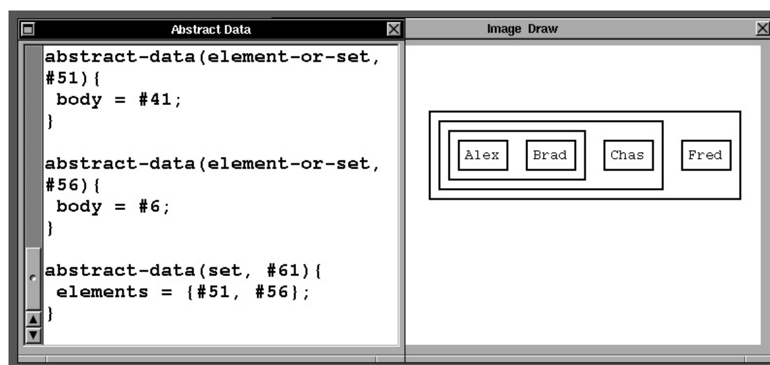


Figure A.13: IMAGE — set diagram screenshot.

In the IMAGE system, the programmer should specify the type definitions of ASR. These are used for generation of a series of ASR data examples. By presenting them one by one to the programmer, the system navigates and helps programmers to provide example pictures to the system.

Using Constraint Hierarchy The visual parser and the visualization engine of IMAGE make extensive use of the constraint hierarchy mechanism[12]. Briefly, each constraint in the constraint hierarchy system has a strength assigned to it. The constraint solver tries to satisfy as many stronger constraints as possible. Therefore, the solver can naturally handle over-constrained systems. Under-constrained systems can be easily converted to over-constrained systems by adding *stay* constraints that preserve the current value of each variable, so they can also be handled in the constraint hierarchy mechanism.

In the IMAGE system, this mechanism is utilized mainly in the revision of visual mapping rules. That is, the corrections of visual mapping rules can be achieved simply by adding new stronger constraints that arrange graphical objects.

A.3 Summary

This chapter described our approaches for interactively generating visual mapping rules. The TRIP3 and IMAGE systems, and examples of their use, were also described. In TRIP3, the programmer provides a pair of examples to the system, i.e., ASR data and the corresponding picture. TRIP3 generates visual mapping rules by generalizing the instances in the example utilizing various heuristics. It is also possible to generate visual mapping rules for recursive data structures by providing examples for the terminal case and the non-terminal case. In the IMAGE system, the programmer can provide multiple example pictures to the system. The system presents a series of example ASR data and sample visualizations to the programmer, who then corrects them interactively, and they are fed back to the system to improve the visual mapping rules.

The approaches taken in these systems have led to proposed solutions to the problems of TRIP2 and TRIP2a. Using these systems, the programmer does not need to write textual visual mapping rules. In addition, programmers usually have to provide slightly different visual mapping rules and inverse visual mapping rules. These are unified in the IMAGE system. The approach described in this chapter should also be beneficial for PBE systems other than the TRIP systems.

Appendix B

TRIP2a/3D

The TRIP2a system described in Chapter 5 displays only two-dimensional figures. There are cases in which three-dimensional representation would be more appropriate. For example, using a three-dimensional view is more natural to represent quad-trees that handle two-dimensional areas (Figure B.12). This section introduces TRIP2a/3D, which is a system for creating 3D animations based on our model, and also shows several example animations constructed using TRIP2a/3D.

B.1 How to Make an Animation

The outline of making an animation with TRIP2a/3D is presented below.

1. Write an application program to be visualized, which is both the most important and the most difficult task in the process.
2. Design an animation. That is, think how to visualize the execution or the algorithm of an application program.
3. Define an ASR for this animation. The ASR should contain sufficient information to display the target animation. See Section B.3.
4. Write a visual mapping rule, e.g., `aRule.kl1`, that translates the defined ASR to VSR that represents the picture designed for this animation. See Section B.2 & Section B.4.
5. Compile the visual mapping rule and make a translator.
   ```
   % cd ~/trip2a/beta
   % make PROGRAM=aRule
   ```
   This compiles `aRule.kl1` and makes the translator `aRule`.
6. Insert some code into the application program to output ASR data during its execution to stdio. For example, when writing a program in C, insert `printf` appropriately.
7. Test the translator and view the generated animation.
   (a) Execute your application and get a log (ASR data).
   ```
   % application > aLogFile.asr
   ```
   (b) Pass the log file to the translator.
   ```
   % aRule < aLogFile.asr > anAnimation.dPR
   ```
   (c) View the animation.
   ```
   % aViewer anAnimation.dPR
   ```

B.2 How to Write a Visual Mapping Rule to Make an Animation

A visual mapping rule specifies how application data should be visualized as an animation. In our *bi-directional translation model*, this is expressed as mappings between ASR and VSR. Here, we describe how to write a mapping rule to make an animation.

B.2.1 Visual Mapping Rules

Currently, we use *KLIC*, a KL1 to C compiler developed at ICOT, to make translators that map ASR data to VSR data. A visual mapping rule is a KL1 module named `vmr`, which is compiled with several other KL1 modules to make a translator.

An example of a mapping rule is shown in Figure B.1.

```
:- module vmr.

rule(default, Result)  :-  Result = [].

rule(towers(A, B, C), Result)  :-
    Result = [
          cylinder(a, 5,  5, [material([diffuse(0.8,0.8,0.5)])]),
          cylinder(b, 10, 5, [material([diffuse(0.8,0.8,0.5)])]),
          cylinder(c, 15, 5, [material([diffuse(0.8,0.8,0.5)])]),
          cylinder(d, 20, 5, [material([diffuse(0.8,0.8,0.5)])]),
          cylinder(e, 25, 5, [material([diffuse(0.8,0.8,0.5)])]),
          box(x, 50, 5, 50, [material([diffuse(0.9,0.5,0.5)])]),
          box(y, 50, 5, 50, [material([diffuse(0.9,0.5,0.5)])]),
          box(z, 50, 5, 50, [material([diffuse(0.9,0.5,0.5)])]),
          x_parallel([x,y,z],[]),
          x_relative([x,y,z],50,[]),
          y_parallel(A, []),
          y_parallel(B, []),
          y_parallel(C, []),
          y_relative(A, 5, []),
          y_relative(B, 5, []),
          y_relative(C, 5, []),
          place(x, 100, 100, 100, [])].

rule(go(A, _), Result)  :-
    Result = [move(A, [circuitous(v1(0, -100, 0), v2(0, 100, 0))])].
```

Figure B.1: A mapping rule example.

The module `vmr` consists of a number of predicates named `rule/2`. The first argument of the predicate `rule` is a term defined in ASR[1], and the corresponding VSR data are listed in the list unified with `Result`. The meaning of this `rule` is that the term defined in the first argument of `rule/2` is visualized as a picture described by the VSR predicates (graphical objects and relations) listed in `Result`.

The module `vmr` consists of a number of predicates named `rule/2`. The first argument of the predicate `rule` is a term defined in ASR, and the corresponding VSR data are listed in the list unified with `Result`. The meaning of this `rule` is that the term defined in the first argument of `rule/2` is visualized as a picture described by the VSR predicates (graphical objects and relations) listed in `Result`.

[1] In fact, mapping rules *are* the definition of ASR. Applications must output ASR data that can be interpreted by the mapping rule.

Using these `rule/2`s in the `vmr` module, all ASR data are translated to VSR. After this translation, graphical constraints among graphical objects in VSR are solved together to determine the coordinates of graphical objects. To obtain the solutions of constraints properly, appropriate graphical constraints must be provided so that the system is neither over-constrained nor under- constrained. Over/under-constrained systems will cause an error and cannot output a picture.

Figure B.1 is a mapping rule for visualizing the tower of Hanoi. The first `rule` is a special one that is always applied once when translating a set of ASR data into a picture. In this example, no default VSR data are generated.

The second rule defines how towers are visualized. The ASR data of the tower is:

```
towers(A, B, C)
```

where each argument (A, B, and C) contains a list of disks at the three possible positions. In this rule, each disk is mapped to a cylinder that has a name (`a` to `e`). The three positions in which disks can be placed are represented as three boxes (`x` to `z`)[2]. These disks and boxes are constrained by nine graphical relations. Briefly, each tower represented by the three lists (at the arguments of `towers`) is placed regularly parallel to the y-axis.

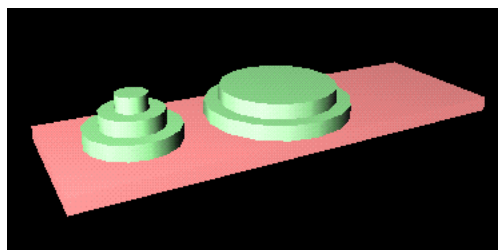


Figure B.2: 3D tower of Hanoi.

The third rule defines transition mapping. A transition mapping rule translates *an abstract operation* on ASR to a transition operation on VSR. This rule maps `go(A, B)` to

```
move(A, [circuitous(v1(0,-100, 0),v2(0, 100, 0))])
```

, which means that the object `A` should move circularly to its destination.

In this example, VSR predicates are enumerated directly in the list. However, `rule/2` is only a predicate that unifies VSR terms to `Result`. Thus, it is possible to write a rule that performs more complex computations to generate VSR predicates.

B.2.2 The Naming of Objects

Each graphical object must have a name, which is specified at the first argument of predicates for graphical objects in VSR. Ground terms such as symbols, lists, and structures can be used as names for graphical objects. However, numbers cannot be used as names. Some examples are presented below:

```
a, b, c, x1, y5, z10
l(10), edge(a,b)
[a,b,c], node(a, [b,c,d])
```

The names of objects are used for two purposes:

[2]In Figure B.2, these three boxes are placed close together so that they look like a board.

Graphical relations As mentioned above, graphical objects and graphical relations define pictures. Graphical relations refer to graphical objects by their names in their arguments. An example is presented below:

```
% three boxes named a, b, and c
box(a, 10, 10, 10, []).
box(b, 10, 10, 10, []).
box(c, 10, 10, 10, []).
% put three boxes in a line
x-parallel([a,b,c], []).
x-relative([a,b,c], 10 ,[])
```

Animations TRIP2a/3D creates animations by comparing each pair of successive pictures. How objects are changed in the transition is determined by comparing the attributes[3] of graphical objects in each picture.

To determine which object in one picture corresponds to which object in another picture, the name of the object is used. That is, if the names of two objects in two pictures are the same, it is assumed that these two objects are identical.

Therefore, to animate objects properly, the programmer should maintain consistent names of objects across the pictures. This usually requires application programs to maintain some information for the names of objects.

For example, consider an animation of a sorting algorithm, where numbers to be sorted are represented as bars (Figure B.3). To depict the swapping of numbers as the movement of bars, each bar must have a corresponding value as its name (Figure B.3(1)). If the index of the array is used as a name (Figure B.3(2)), the bar does not move but is shrunk/enlarged in the transition.

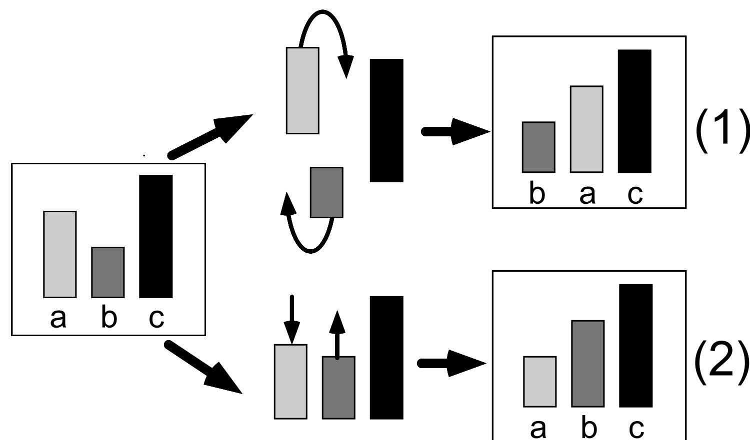


Figure B.3: Two ways of naming objects.

B.3 ASR Data Representation

Abstract Structure Representation (ASR) is the input for the TRIP2a/3D system. To animate the execution of an application, applications must output their internal data and operations in the form

[3]such as the positions and the sizes

of ASR. This section describes the format of ASR.

B.3.1 Model

TRIP2a/3D reads a series of ASR data and operations on ASR, and converts them into an animation.

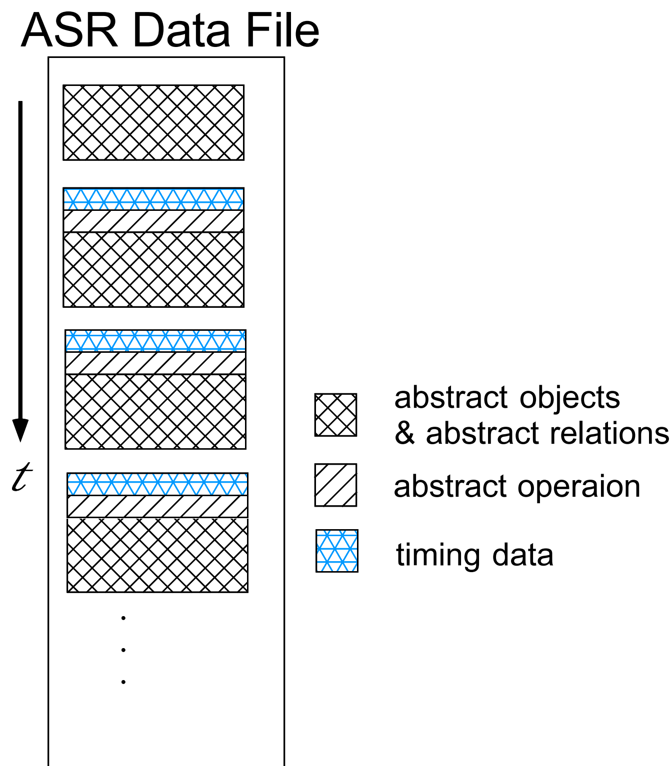


Figure B.4: ASR data file.

The ASR data consist of the following:

Time The time at which the operation is executed.

Internal Application Data The data in an application to be visualized. The ASR data represent internal application data corresponding to a picture.

Operations on ASR An operation on ASR represents an operation that changes the internal state of an application during its execution. This operation corresponds to the movement or transformation of graphical objects in an animation.

The system creates an animation by comparing pictures generated from every pair of consecutive ASR data sets. The first set of ASR data is used to generate the initial picture of an animation, so it should not contain an operation or timing information.

The order of each term in a set of ASR data is not important except that `time/1`. `time/1` must be the first term in each set.

B.3.2 Syntax

```
asrs     : asr
```

```
        | asr asrs
        ;

asr     : terms end_term
        ;

terms   : term
        | terms
        ;

term    : <terms in Prolog/klic> '.' /* shoudn't includes variables */
        ;

end_term: "end_of_state."
```

B.3.3 An Example

```
object(a).
object(b).
place(a, 10, 10, 0).
place(b, 20, 10, 0).
object(c).
object(d).
place(c, 10, 20, 0).
place(d, 10, 30, 0).
end_of_state.

time(1.0).
object(a).
object(b).
place(a, 30, 10, 0).
place(b, 10, 20, 0).
object(c).
object(d).
place(c, 20, 10, 0).
place(d, 30, 10, 0).
end_of_state.

time(2.0).
object(a).
object(b).
place(a, 30, 30, 0).
place(b, 10, 10, 0).
object(c).
object(d).
place(c, 10, 20, 0).
place(d, 10, 30, 0).
end_of_state.

time(3.0).
object(a).
object(b).
place(a, 10, 30, 0).
place(b, 20, 20, 0).
object(c).
object(d).
```

```
place(c, 10, 30, 0).
place(d, 20, 10, 0).
end_of_state.

time(4.0)
object(a).
object(b).
place(a, 10, 10, 0).
place(b, 10, 10, 0).
object(c).
object(d).
place(c, 10, 10, 0).
place(d, 10, 10, 0).

end_of_state.
```

B.4 VSR Specification

To handle three-dimensional animations, Visual Structure Representation (VSR), which represents the structure of pictures, is extended to handle three-dimensional pictures. The programmer writes a visual mapping rule that maps abstract data to three-dimensional graphical objects and constraints. The following are currently available graphical objects and constraints.

B.4.1 Graphical Objects

These predicates represent 3D objects. Currently, their sizes, lengths, and colors can be specified, but not their directions. Each object has its own coordinates (x, y, and z), which are calculated from the graphical relations by which they are constrained.

sphere(Name, Radius, Modes) :

`Name : term`
`Radius : integer`
`Modes : modes`
A sphere with radius = `Radius`.

box(Name, Width, Height, Depth, Modes) :

`Name : term`
`Width, Height, Depth : integer`
`Modes : modes`
A box with width = `Width`, height = `Height`, depth = `Depth`.

cylinder(Name, Radius, Height, Modes) :

`Name : term`
`Radius, Height : integer`
`Modes : modes`
A cylinder with radius = `Radius`, height = `Height`.

cone(Name, Radius, Height, Modes) :

`Name : term`
`Radius, Height : integer`
`Modes : modes`
A cone with radius = `Radius`, height = `Height`.

line(Name, Radius, Modes) :

`Name : term`
`Radius : integer`
`Modes : modes`

A line with thickness Radius. The length of a line cannot be specified, as its two end points are determined by a `connect` relation. If it is necessary to specify length, `cylinder` should be used instead.

word(Name, Text, Modes) :

`Name : term`
`Text : string`
`Modes : modes`

The string object `Text` is displayed on the screen in 2D.

Modes The last argument of the predicates for graphical objects is Modes, which aims to specify various attributes of objects. At present, only the color of objects can be specified. The method of specifying colors is based on OpenInventor. The following values can be specified in a list at the argument of `material/1`.

```
ambient(R, G, B)
diffuse(R, G, B)
specular(R, G, B)
emissive(R, G, B)
shininess(V)
transparency(V)
```

Please refer to the OpenInventor Manual[126] for the meanings of these terms. Note that the viewer that uses Amulet ignores the type of values; i.e., only RGB values are important. Shininess and transparency values are also ignored.

B.4.2 Graphical Relations

Here, assume that `NameList = [obj1, obj2, obj3, ...]`. Currently, `Modes` are ignored in these graphical relations, but exist for future improvements.

x_parallel(NameList, Modes) :

`obj1.y = obj2.y, obj2.y = obj3.y, ...`
`obj1.z = obj2.z, obj2.z = obj3.z, ...`

Graphical objects listed in the `NameList` are arranged parallel to the x-axis.

y_parallel(NameList, Modes) :

`obj1.z = obj2.z, obj2.z = obj3.z, ...`
`obj1.x = obj2.x, obj2.x = obj3.x, ...`

Graphical objects listed in the `NameList` are arranged parallel to the y-axis.

z_parallel(NameList, Modes) :

`obj1.x = obj2.x, obj2.x = obj3.x, ...`
`obj1.y = obj2.y, obj2.y = obj3.y, ...`

Graphical objects listed in the `NameList` are arranged parallel to the z-axis.

x_relative(NameList, Gap, Modes) :

`obj1.x + Gap = obj2.x, obj2.x + Gap = obj3.x, ...`

Graphical objects listed in the `NameList` are placed regularly at intervals of `Gap` along the x-axis.

y_relative(NameList, Gap, Modes) :

`obj1.y + Gap  = obj2.y, obj2.y + Gap  = obj3.y, ...`

Graphical objects listed in the `NameList` are placed regularly at intervals of `Gap` along the y-axis.

z_relative(NameList, Gap, Modes) :

`obj1.z + Gap = obj2.z, obj2.z + Gap = obj3.z, ...`

Graphical objects listed in the `NameList` are placed regularly at intervals of `Gap` along the z-axis.

place(Name, X, Y, Z, Modes) :

`Name.x = X, Name.y = Y, Name.z = Z`

A graphical object is placed at `X, Y, Z`.

x_average(Name, NameList, Modes) :

`(obj1.x + obj2.x + obj3.x + ...)/N = Name.x`

An object named `Name` is placed at the average coordinates on the x-axis of the objects in the `NameList`.

y_average(Name, NameList, Modes) :

`(obj1.y + obj2.y + obj3.y + ...)/N = Name.y`

An object named `Name` is placed at the average coordinates on the y-axis of the objects in the `NameList`.

z_average(Name, NameList, Modes) :

`(obj1.z + obj2.z + obj3.z + ...)/N = Name.z`

An object named `Name` is placed at the average coordinates on the z-axis of the objects in the `NameList`.

xy_circular(Name, NameList, Radius, Modes) :

```
obj1.x = Name.x + Radius*cos(Theta1),
obj2.x = Name.x + Radius*cos(Theta2),...
obj1.y = Name.y + Radius*cos(Theta1),
obj2.y = Name.y + Radius*cos(Theta2),...
```

The objects in the `NameList` are placed circularly around the object `Name` in the xy-plane.

yz_circular(Name, NameList, Radius, Modes) :

```
obj1.y = Name.y + Radius*cos(Theta1),
obj2.y = Name.y + Radius*cos(Theta2),...
obj1.z = Name.z + Radius*cos(Theta1),
obj2.z = Name.z + Radius*cos(Theta2),...
```

The objects in the `NameList` are placed circularly around the object `Name` in the yz-plane.

zx_circular(Name, NameList, Radius, Modes) :

```
obj1.z = Name.z + Radius*cos(Theta1),
obj2.z = Name.z + Radius*cos(Theta2),...
```

```
obj1.x = Name.x + Radius*cos(Theta1),
obj2.x = Name.x + Radius*cos(Theta2),...
```

The objects in the `NameList` are placed circularly around the object `Name` in the zx-plane.

B.4.3 Transitional Operations

Transitional operations are used for specifying the movements of objects in an animation. If not specified, the default operation that moves the object in a straight line is used.

move(Name, Modes) :
This predicate is used to express how objects should be moved in an animation. The command is specified in `Modes`, which is a list that contains one of the following types:

via(N, PList) :
This specifies the relay points in this transition. `PList` is a list of $N - 1$ points. For example,

```
move(obj1, via(3,[[0,0,0],[100,100,100]]))
```

means that the object `obj1` moves via the two specified points ((0,0,0) and (100,100,100)).

circuitous(v1(V1x, V1y, V1z), v2(V2x, V2y, V2z)) :
This specifies the tangent vectors at the start and the end of movement of the object. `v1/3` is the tangent vector at the starting position, and `v2/3` is that at the ending position. The path of the object is determined by the start/end positions and the two tangent vectors. To calculate the path, the Hermite form of the cubic polynomial curve is used [40].

from(Obj, ID), to(Obj, ID) :
These are transitional operations that specify asynchronous movements. See Chapter 5.5 for details.

B.4.4 An Example

Figure B.5 shows a set of VSR data representing the tower of Hanoi shown in Figure B.2. In fact, the programmer has no need to deal with VSR data directly. Instead, the programmer can write only VSR predicates in the mapping rule to generate VSR data from ASR data.

B.5 Examples

B.5.1 N-Queen Problem

Figure B.6 shows two screenshots from the three-dimensional animation of N-Queen problem solving. A queen is represented as a yellow box. Queens to be placed are initially arranged at the left, and are placed on the board one by one. Each placement of a queen is shown as the movement of a queen from the left position to the position on the board. After a queen is put on the board, the position covered by the queen is changed to red. In Figure B.6(a), one queen is placed on the board, so the same horizontal, vertical, and diagonal row are red. In Figure B.6(b), three queens are put on the board, and the positions that they cover are red.

The boxes representing queens are distorted when they are moved for placement on the board. Figure B.7 shows how a queen is distorted when it is moving. The box is stretched in the direction of movement in proportion to its acceleration.

```
% graphical objects
cylinder(a, 5,  5, [material([diffuse(0.8,0.8,0.5)])]).
cylinder(b, 10, 5, [material([diffuse(0.8,0.8,0.5)])]).
cylinder(c, 15, 5, [material([diffuse(0.8,0.8,0.5)])]).
cylinder(d, 20, 5, [material([diffuse(0.8,0.8,0.5)])]).
cylinder(e, 25, 5, [material([diffuse(0.8,0.8,0.5)])]).
box(x, 50, 5, 50, [material([diffuse(0.9,0.5,0.5)])]).
box(y, 50, 5, 50, [material([diffuse(0.9,0.5,0.5)])]).
box(z, 50, 5, 50, [material([diffuse(0.9,0.5,0.5)])]).
% graphical relations
x_parallel([x,y,z],[]).
x_relative([x,y,z],50,[]).
y_parallel([a,b,c,x], []).
y_parallel([d,e,y], []).
y_parallel([z], []).
y_relative([a,b,c,x], 5, []).
y_relative([d,e,y], 5, []).
y_relative([z], 5, []).
place(x, 100, 100, 100, []).
% transitional operations
move(d, [circuitous(v1(0, -100, 0), v2(0, 100, 0))]).
```

Figure B.5: VSR data example.

B.5.2 The Tower of Hanoi

Figure B.9 shows the three-dimensional animations of the tower of Hanoi. In this animation, the movement of a plate, although represented as a box, is exaggerated, which makes the animation more vivid. In the same way as in the N-Queen animation, the moving box is stretched in the direction of its movement in proportion to its acceleration. The visual mapping rule set for this animation is shown in Figure B.1.

B.5.3 N-Body Simulation

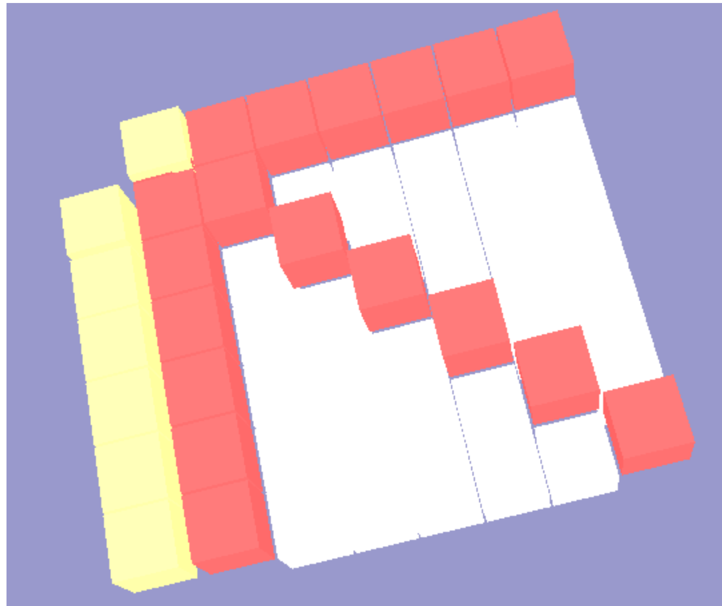
The N-Body problem is to simulate the behavior of N particles interacting with each other through a long-range force such as gravity or Coulombic force. Figure B.12 shows a screenshot from the animation that depicts the execution of a two-dimensional N-Body simulation program.

In the simulation, the program uses a quad-tree for handling the particles. The entire two-dimensional space is regarded as a large square. The square is recursively divided into four parts by dividing vertically and horizontally until each part has only one particle.

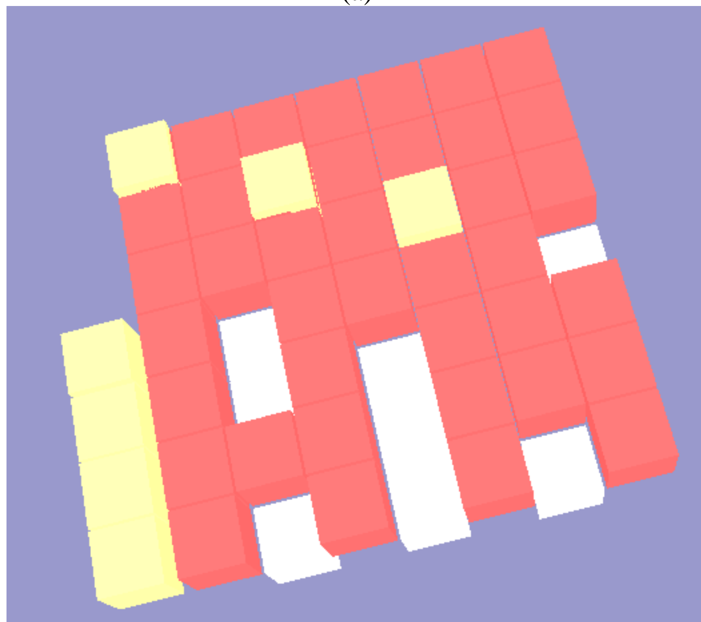
The simulation executes multiple processes, each of which has a quad-tree that represents particles in the two-dimensional space. In Figure B.12, two processes are executed in the simulation. The left quad-tree is managed by one process, and the right quad-tree is managed by another.

However, each process does not have all the data from the tree. For example, the process corresponding to the right tree has the data represented as blue square nodes in the right tree. It does not have the data represented as blue semitransparent nodes. The data from the left tree are represented as red opaque nodes, and the process managing the left tree does not own the semitransparent red nodes in the left tree.

When a process notices that it does not have some part of the tree, it asks the other process to send data. This is represented as an animation consisting of sending a message represented as a red sphere (Figure B.12(a)(b)), and receiving of the data represented as a yellow square (Figure B.12(c)(d)(e)). The received data are cached during the process. In Figure B.12(f), the cached



(a)



(b)

Figure B.6: 7 queen problem.

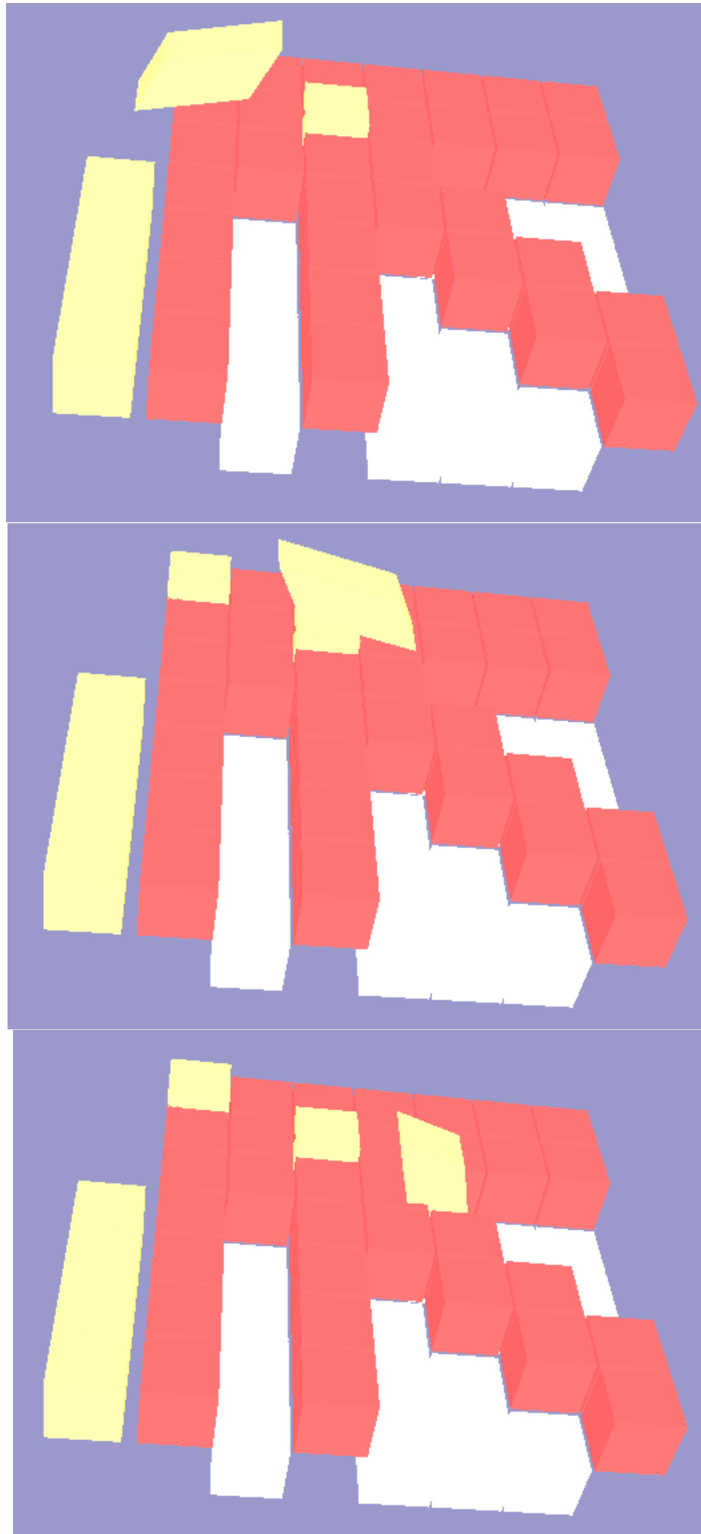


Figure B.7: 7 queen problem — distortion technique.

```
:- module vmr.
rule(default, Result) :-
  Result = [
    x_relative([b(1,1),b(1,2),b(1,3),b(1,4),b(1,5),b(1,6),b(1,7)],52,[]),
    x_relative([b(2,1),b(2,2),b(2,3),b(2,4),b(2,5),b(2,6),b(2,7)],52,[]),
    x_relative([b(3,1),b(3,2),b(3,3),b(3,4),b(3,5),b(3,6),b(3,7)],52,[]),
    x_relative([b(4,1),b(4,2),b(4,3),b(4,4),b(4,5),b(4,6),b(4,7)],52,[]),
    x_relative([b(5,1),b(5,2),b(5,3),b(5,4),b(5,5),b(5,6),b(5,7)],52,[]),
    x_relative([b(6,1),b(6,2),b(6,3),b(6,4),b(6,5),b(6,6),b(6,7)],52,[]),
    x_relative([b(7,1),b(7,2),b(7,3),b(7,4),b(7,5),b(7,6),b(7,7)],52,[]),
    x_parallel([b(1,1),b(1,2),b(1,3),b(1,4),b(1,5),b(1,6),b(1,7)],[]),
    x_parallel([b(2,1),b(2,2),b(2,3),b(2,4),b(2,5),b(2,6),b(2,7)],[]),
    x_parallel([b(3,1),b(3,2),b(3,3),b(3,4),b(3,5),b(3,6),b(3,7)],[]),
    x_parallel([b(4,1),b(4,2),b(4,3),b(4,4),b(4,5),b(4,6),b(4,7)],[]),
    x_parallel([b(5,1),b(5,2),b(5,3),b(5,4),b(5,5),b(5,6),b(5,7)],[]),
    x_parallel([b(6,1),b(6,2),b(6,3),b(6,4),b(6,5),b(6,6),b(6,7)],[]),
    x_parallel([b(7,1),b(7,2),b(7,3),b(7,4),b(7,5),b(7,6),b(7,7)],[]),
    y_relative([b(1,1),b(2,1),b(3,1),b(4,1),b(5,1),b(6,1),b(7,1)],52,[]),
    y_parallel([b(1,1),b(2,1),b(3,1),b(4,1),b(5,1),b(6,1),b(7,1)],[]),
    place(b(1,1), 0, 0, 0, [])
  ].
rule(obj(X), Result) :-
  Result = [box(X, 50, 50, 50, [material([ambient(0.8,0.8,0.3)])])].
rule(at(Obj, X, 0), Result) :-
  Result = [x_relative([Obj, b(X,1)], 60, []),
            y_relative([Obj, b(X,1)], 0, []),
            z_relative([Obj, b(X,1)], 0, [])].
otherwise.
rule(at(Obj, X, Y), Result) :-
  Result = [z_parallel([Obj,b(X,Y)],[]),z_relative([Obj, b(X,Y)],12,[])].
rule(base(X,Y,ok), Result) :-
  Result = [box(b(X,Y), 50, 50, 5, [material([diffuse(0.8,0.8,0.8)])])].
rule(base(X,Y,ng), Result) :-
  Result = [box(b(X,Y), 50, 50, 50, [material([diffuse(0.8,0.2,0.2)])])].
rule(move(X,Y), Result) :- Result = [move(X,[Y])].
```

Figure B.8: A visual mapping rule set for the seven queen problem solving animation.

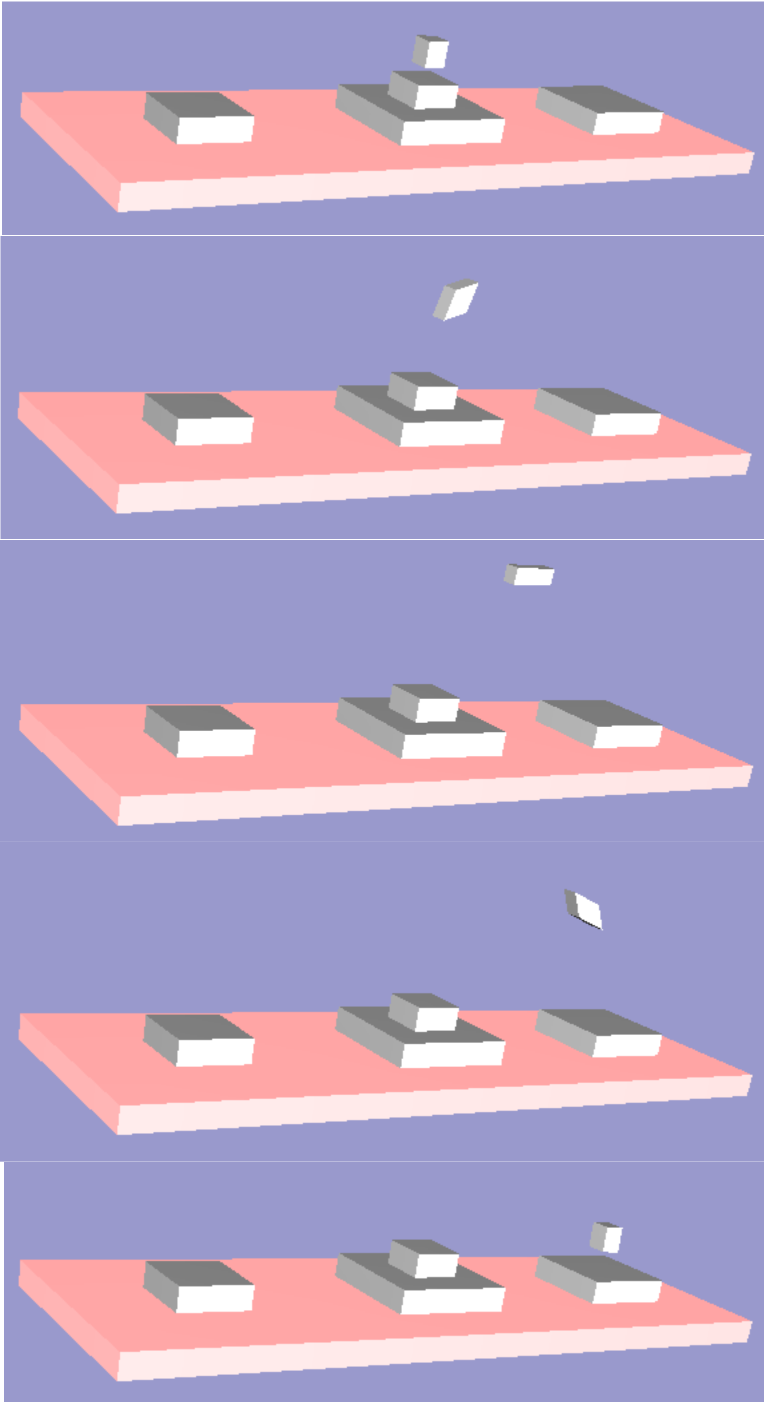


Figure B.9: The 3D tower of Hanoi animation.

data are represented as the red square node, which is the same color as the node that sent the data.

The animation proceeds by showing where in the tree the calculation is executing, and also showing the acquisition of missing data from the other process. Where the calculation is being executed is represented as the position of the white sphere node. In Figure B.12(a) ∼ (e), the white sphere is at the node of depth=1 in the right tree that sent the message and received the missing data.

Figure B.11 shows part of the visual mapping rule set for Figure B.12. This mapping rule set maps the ASR data in Figure B.10 to the animation in Figure B.12.

```
% The Initial State
world([space0, space1]).
space(space1,1, t1_0,
        [tree(t1_0, [t1_4, t1_3, t1_2, t1_1]),
         tree(t1_17, [t1_72, t1_71, t1_70, t1_69]),
         tree(t1_4, [t1_20, t1_19, t1_18, t1_17]),
         tree(t1_1, [t1_8, t1_7, t1_6, t1_5])]).
empty(t1_0,0,0). fill(t1_17,1,2). fill(t1_72,1,3).
...
space(space0,0, t0_0,
        [tree(t0_0, [t0_4, t0_3, t0_2, t0_1]),
         tree(t0_17, [t0_72, t0_71, t0_70, t0_69]),
         tree(t0_4, [t0_20, t0_19, t0_18, t0_17]),
         tree(t0_1, [t0_8, t0_7, t0_6, t0_5])]).
fill(t0_0,0,0). empty(t0_17,1,2). empty(t0_72,1,3).
...
end_of_state.
% The Second State
world([space0, space1]).
space(space1,1, t1_0,
        [tree(t1_0, [t1_4, t1_3, t1_2, t1_1]),
         tree(t1_17, [t1_72, t1_71, t1_70, t1_69]),
         tree(t1_4, [t1_20, t1_19, t1_18, t1_17]),
         tree(t1_1, [t1_8, t1_7, t1_6, t1_5])]).
...
```

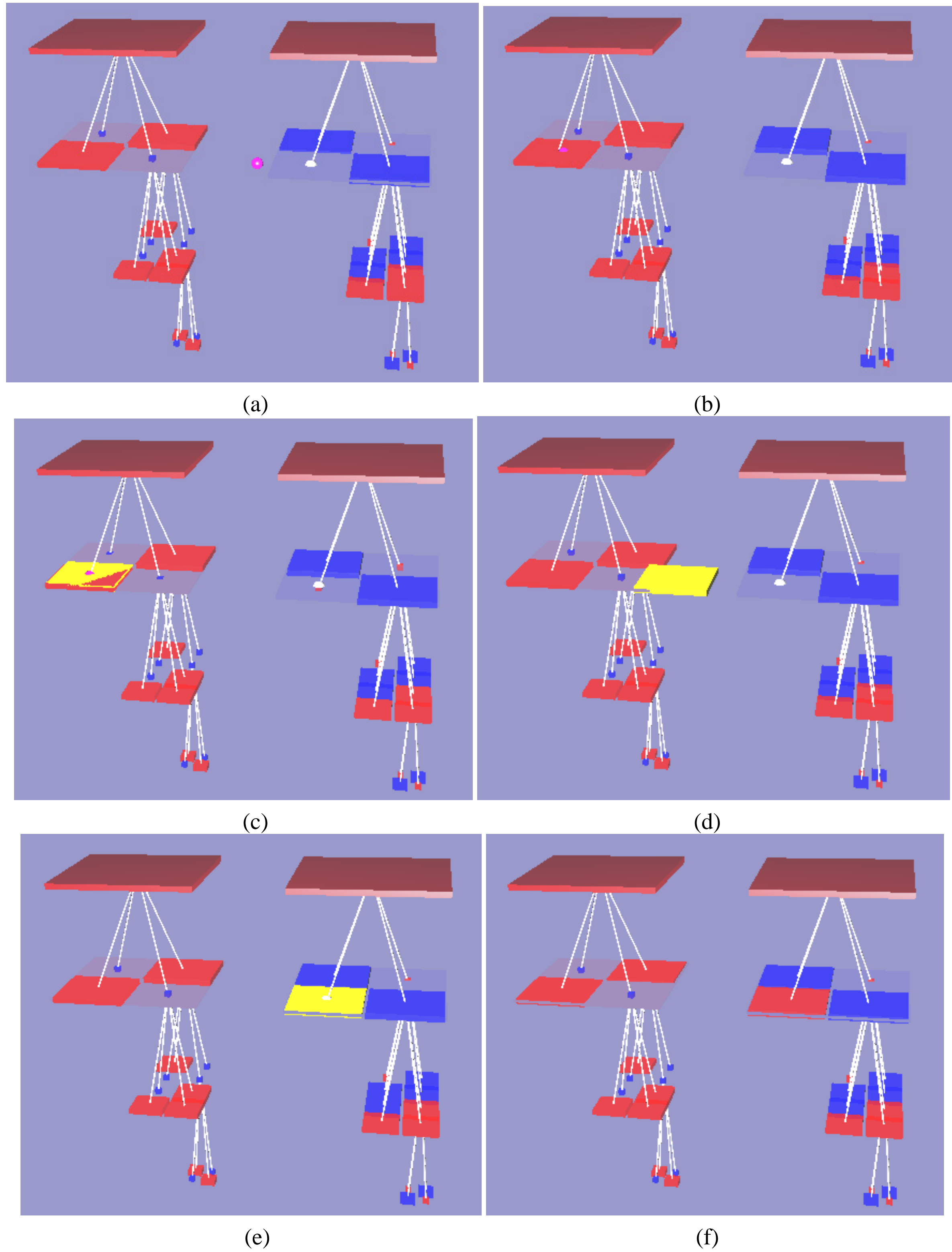
Figure B.10: Input ASR data list for N-body animation.

B.5.4 Memory Management

Figure B.13 shows a screenshot of an animation that depicts the behavior of a distributed shared memory system in the COS operating system. See reference [39] for more details.

```
% Visual mapping rules for N-Body algorithm using quad-tree
:- module vmr.
rule(default, R) :- R = [].
rule(world(SpaceList), Result) :-
    SpaceList = [Head|_],
    Result = [ x_relative(SpaceList, 300, []),
               x_parallel(SpaceList, []),
               place(Head, 0, 0, 0, [])].
rule(space(SpaceID, NodeNum, Root, Trees), Result) :- true |
    treemap(Root, Trees, 128, R),
    getColor(NodeNum, Color),
    Result = [box(SpaceID, 256, 256, 400,
                [material([transparency(0.7), Color])]),
              x_relative([SpaceID, Root], 0, []),
              y_relative([SpaceID, Root], 0, []),
              z_relative([SpaceID, Root], 200, []) | R].
rule(fill(ID, ColorID, Level), Result) :-
    getColor(ColorID, Color),
    getSize(Level, Size),
    EColor = emissive(0.0,0.0,0.0),
    TColor = transparency(0.0),
    Result = [box(ID, Size, Size, 3, [material([EColor,TColor,Color])])].
rule(empty(ID, ColorID, Level), Result) :-
    getColor(ColorID, Color),
    getSize(Level, Size),
    EColor = emissive(0.0,0.0,0.0),
    TColor = transparency(0.7),
    Result = [box(ID, Size, Size, 3, [material([EColor,TColor,Color])])].
rule(calc(ID, ColorID, Level), Result) :-
    getColor(ColorID, Color),
    getEColor(ColorID, EColor),
    getSize(Level, Size),
    TColor = transparency(0.0),
    Result = [box(ID, Size, Size, 3, [material([EColor,TColor,Color])])].
rule(calc_token(ID), Result) :-
    Result = [sphere(calcToken, 10, []),
              x_relative([ID, calcToken], 0, []),
              y_relative([ID, calcToken], 0, []),
              z_relative([ID, calcToken], 0, [])].
rule(cache_token(ID), Result) :-
    Result = [sphere(cacheToken, 10, [material([diffuse(1.0,0.0,1.0)])]),
              x_relative([ID, cacheToken], 0, []),
              y_relative([ID, cacheToken], 0, []),
              z_relative([ID, cacheToken], 0, [])].
rule(fill_token(ID, Level), Result) :-
    getSize(Level, Size),
    Result = [box(cacheReturn, Size, Size, 3,
                [material([diffuse(1.0,0.0,1.0)])]),
              x_relative([ID, cacheReturn], 0, []),
              y_relative([ID, cacheReturn], 0, []),
              z_relative([ID, cacheReturn], 0, [])].
...omitted...
```

Figure B.11: An example of visual mapping rules for 3D visualization: quad-tree (excerpt).



(a) (b)

(c) (d)

(e) (f)

Figure B.12: An example of 3D visualization: quad-tree.

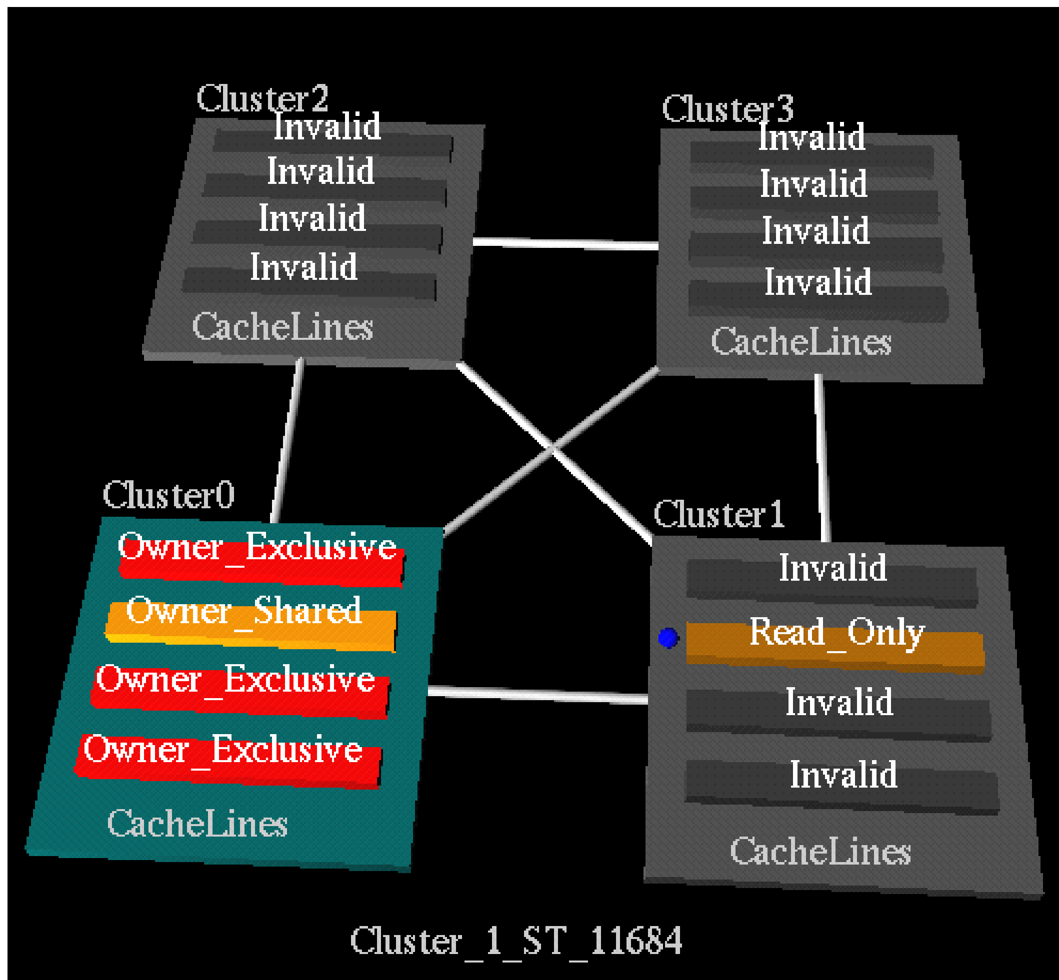


Figure B.13: An example of 3D visualization: cache.

Appendix C

Examples of Mapping Rules for TRIP2

C.1 Small Graph Editor

```
%% Mapping Rules for a Small Graph Editor
% Sample data
%   edge(a,b).      edge(b,c).
%   edge(a,c).      edge(a,d).
%   node(a).        node(b).
%   node(c).        node(d).
% Visual mapping driver
    objectmap([o]).
    o :- node(X), nodemap(X), fail.        % iteration that maps all the nodes
    relationmap([r]).
    r :- edge(A, B), edgemap(A, B), fail.% iteration that maps all the edges
% Visual mapping rules
    nodemap(X) :-      % a node is mapped to a box which contains a label
        circle(X, 15, []), label(l(X), X, []), contain(X, l(X), 0, []).
    edgemap(A, B) :- % an edge is mapped to a connection between specified nodes
        connect(A, B, center, center, []), adjacent(A, B, 1).
% Inverse visual mapping driver
    invomap([io]).
    invrmap([ir]).
    io :- % iteration that searches and outputs all the nodes
      invnodemap(X), asr(node(X)), fail.
    ir :- % iteration that searches and outputs all the edges
      invedgemap(A, B), asr(edge(A, B)), fail.
% Inverse visual mapping rules
    invnodemap(X) :-     % searches for a circle that contains a label
       circle(A, _, _), label(B, X, _), contain(A, B, _, _).
    invedgemap(A, X) :- % searches for a connection, which corresponds to an edge
       connect(A, X, _, _, _).
```

C.2 Othello Game Application

```
objectmap([omap]).
relationmap([rmap]).

omap :-
        board(L),
        omap1(1, 1, L).

omap1(_, _, [[]]).
omap1(N, _, [[]|R]) :-
        N1 is N+1,
        omap1(N1, 1, R).
omap1(N, M, [[BW|L]|R]) :-
        stone([N,M], BW),
        M1 is M+1,
        omap1(N, M1, [L|R]).

stone([N,M], e) :-
        box([N,M], 30, 30, [bound,blue]).
stone([N,M], b) :-
        box([N,M],30,30,[bound,blue]),
        circle([N,M,b],15,[fill]),
```

```
        contain([N,M],[N,M,b],0,[]).
stone([N,M], w) :-
        box([N,M],30,30,[bound,blue]),
        circle([N,M,w],15,[visible]),
        contain([N,M],[N,M,w],0,[]).

rmap :-
        abolish(boarddata(_)),
        board(L),
        assert(boarddata(L)),
        rmap1(L),
        rmap2(L).

rmap1(L) :- rmap1(L, 1).
rmap1([], _).
rmap1([L|R], M) :-
        objlist(L, M, 1, OL),
        horizontal(OL, []),
        x_order(OL, 0, []),
        M1 is M+1,
        rmap1(R, M1).
rmap2(L) :-
        headlist(L, HL, 1),
        vertical(HL, []),
        y_order(HL, 0, []).

objlist([], _, _, []).
objlist([_|R], M, N, [[M, N]|X]) :-
        N1 is N+1,
        objlist(R, M, N1, X).

headlist([], [], _).
headlist([_|R], [[N,1]|X], N) :-
        N1 is N+1,
        headlist(R, X, N1).

invomap([]).
invrmap([irmap]).
irmap :-
        irmap2(L),
        irmap1(L, X),
        conv(X,R),
        rev(R,A),
        asr(board(A)).

irmap2(L) :- y_order(L, _, _).

irmap1([H|R],[[H|W]|X]) :-
        x_order([H|W], _, _),
        irmap1(R, X).
irmap1([],[]).

conv([],[]).
conv([A|B],[X|Y]) :-
        conv_sub(A, X),
        conv(B, Y).
conv_sub([H|T],[I|X]) :-
        (contain(H,C,_,_)
                -> ( circle(C, _, [fill]) -> I = 'b'; I = 'w');
        /* else */
                      I = 'e'),
        conv_sub(T,X).
conv_sub([],[]).

part(X, [A|Y]) :- head(X, [A|Y]); part(X, Y).

head([], _).
head([A|X], [A|Y]) :- head(X, Y).


last_stone(A, B, I, J) :- last_stone(A, B, 1, 1, I, J).

last_stone([A|X], [B|Y], N, M, I, J) :-
        last_stone_sub(A, B, N, M, I, J);
        N1 is N + 1,
        last_stone(X, Y, N1, M, I, J).

last_stone_sub([e|X], [w|Y], N, M, N, M).
```

```
last_stone_sub([e|X], [b|Y], N, M, N, M).
last_stone_sub([_|X], [_|Y], N, M, I, J) :-
        M1 is M + 1,
        last_stone_sub(X, Y, N, M1, I, J).
last_stone_sub([],[],8,8,8,8).

rev1([], []).
rev1([wn|X], [wn|Y]) :-
        bl_w(X) -> rev1_sub1(X, Y) ; X = Y.
rev1([bn|X], [bn|Y]) :-
        w_bl(X) -> rev1_sub2(X, Y) ; X = Y.
rev1([A|R], [A|S]) :- rev1(R, S).

bl_w([b,w|_]).
bl_w([b|X]) :- bl_w(X).

w_bl([w,b|_]).
w_bl([w|X]) :- bl_w(X).

rev1_sub1([b,w|X], [w,w|X]).
rev1_sub1([b|X], [w|Y]) :- rev1_sub1(X, Y).

rev1_sub2([w,b|X], [b,b|X]).
rev1_sub2([w|X], [b|Y]) :- rev1_sub2(X, Y).

rev2(X, Y) :-
        reverse(X, A),
        rev1(A, B),
        reverse(B, Y).

rev_h([],[]).
rev_h([A|B], [C|D]) :-
        rev_h_sub(A, C),
        rev_h(B, D).
rev_h_sub(X, Y) :-
        rev1(X, T),
        rev2(T, Y).

reverse([X], [X]).
reverse([H|R], Z) :-
        reverse(R, S),
        append(S, [H], Z).

append([], X, X).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).

head_stones([[A|X]], [A], [X]).
head_stones([[A|X]|Y], [A|Z], [X|W]) :- head_stones(Y, Z, W).

hori_vert([[]|_], []).
hori_vert(X, [A|Y]) :-
        head_stones(X, A, Z),
        hori_vert(Z, Y).

rev_v(X, Y) :-
        hori_vert(X, A),
        rev_h(A, B),
        hori_vert(B, Y).

stonexy(1, Y, [HB|RB], S) :- stonexy_sub(Y, HB, S).
stonexy(X, Y, [HB|RB], S) :-
        X1 is X - 1,
        stonexy(X1, Y, RB, S).

stonexy_sub(1, [S|_], S).
stonexy_sub(1, _, _) :- !, fail.
stonexy_sub(Y, [H|R], S) :-
        Y1 is Y - 1,
        stonexy_sub(Y1, R, S).


rev_d(X, Y, B, R) :-
        rev_d1(X, Y, B, R1),
        rev_d2(X, Y, R1, R2),
        rev_d3(X, Y, R2, R3),
        rev_d4(X, Y, R3, R).
```

```
rev_d1(X, Y, B, R) :-
        stonexy(X, Y, B, wn), rev_d1_sub1(X, Y, B, R);
        stonexy(X, Y, B, bn), rev_d1_sub2(X, Y, B, R).
rev_d1(_, _, B, B).
rev_d1_sub1(X, Y, B, R) :-
        X1 is X - 1,
        Y1 is Y - 1,
        bl_w_d(X1, Y1, -1, -1, B),
        rev_d_sub1(X1, Y1, -1, -1, B, R).
rev_d1_sub2(X, Y, B, R) :-
        X1 is X - 1,
        Y1 is Y - 1,
        w_bl_d(X1, Y1, -1, -1, B),
        rev_d_sub2(X1, Y1, -1, -1, B, R).

rev_d2(X, Y, B, R) :-
        stonexy(X, Y, B, wn), rev_d2_sub1(X, Y, B, R);
        stonexy(X, Y, B, bn), rev_d2_sub2(X, Y, B, R).
rev_d2(_, _, B, B).
rev_d2_sub1(X, Y, B, R) :-
        X1 is X + 1,
        Y1 is Y - 1,
        bl_w_d(X1, Y1, 1, -1, B),
        rev_d_sub1(X1, Y1, 1, -1, B, R).
rev_d2_sub2(X, Y, B, R) :-
        X1 is X + 1,
        Y1 is Y - 1,
        w_bl_d(X1, Y1, 1, -1, B),
        rev_d_sub2(X1, Y1, 1, -1, B, R).

rev_d3(X, Y, B, R) :-
        stonexy(X, Y, B, wn), rev_d3_sub1(X, Y, B, R);
        stonexy(X, Y, B, bn), rev_d3_sub2(X, Y, B, R).
rev_d3(_, _, B, B).
rev_d3_sub1(X, Y, B, R) :-
        X1 is X - 1,
        Y1 is Y + 1,
        bl_w_d(X1, Y1, -1, 1, B),
        rev_d_sub1(X1, Y1, -1, 1, B, R).
rev_d3_sub2(X, Y, B, R) :-
        X1 is X - 1,
        Y1 is Y + 1,
        w_bl_d(X1, Y1, -1, 1, B),
        rev_d_sub2(X1, Y1, -1, 1, B, R).

rev_d4(X, Y, B, R) :-
        stonexy(X, Y, B, wn), rev_d4_sub1(X, Y, B, R);
        stonexy(X, Y, B, bn), rev_d4_sub2(X, Y, B, R).
rev_d4(_, _, B, B).
rev_d4_sub1(X, Y, B, R) :-
        X1 is X + 1,
        Y1 is Y + 1,
        bl_w_d(X1, Y1, 1, 1, B),
        rev_d_sub1(X1, Y1, 1, 1, B, R).
rev_d4_sub2(X, Y, B, R) :-
        X1 is X + 1,
        Y1 is Y + 1,
        w_bl_d(X1, Y1, 1, 1, B),
        rev_d_sub2(X1, Y1, 1, 1, B, R).

bl_w_d(X, Y, DX, DY, B) :-
        stonexy(X, Y, B, b),
        X1 is X + DX,
        Y1 is Y + DY,
        ( X1 < 1 -> fail;
          ( Y1 < 1 -> fail;
            ( X1 > 8 -> fail;
              ( Y1 > 8 -> fail;
                ( stonexy(X1, Y1, B, w);
                  bl_w_d(X1, Y1, DX, DY, B)))))).

w_bl_d(X, Y, DX, DY, B) :-
        stonexy(X, Y, B, w),
        X1 is X + DX,
        Y1 is Y + DY,
        ( X1 < 1 -> fail;
          ( Y1 < 1 -> fail;
            ( X1 > 8 -> fail;
              ( Y1 > 8 -> fail;
```

```
                ( stonexy(X1, Y1, B, b);
                  w_bl_d(X1, Y1, DX, DY, B)))))).

rev_d_sub1(X, Y, DX, DY, B, B) :- stonexy(X, Y, B, w).
rev_d_sub1(X, Y, DX, DY, B, R) :-
        rev_stonexy(X, Y, w, B, R1),
        X1 is X + DX,
        Y1 is Y + DY,
        rev_d_sub1(X1, Y1, DX, DY, R1, R).

rev_d_sub2(X, Y, DX, DY, B, B) :- stonexy(X, Y, B, b).
rev_d_sub2(X, Y, DX, DY, B, R) :-
        rev_stonexy(X, Y, b, B, R1),
        X1 is X + DX,
        Y1 is Y + DY,
        rev_d_sub2(X1, Y1, DX, DY, R1, R).

rev_stonexy(1, Y, S, [H|B], [R|B]) :- rev_stonexy_sub(Y, S, H, R).
rev_stonexy(X, Y, S, [H|B], [H|R]) :-
        X1 is X - 1,
        rev_stonexy(X1, Y, S, B, R).

rev_stonexy_sub(1, S, [H|R], [S|R]).
rev_stonexy_sub(Y, S, [H|R1], [H|R2]) :-
        Y1 is Y - 1,
        rev_stonexy_sub(Y1, S, R1, R2).

rev(X, Y, A, B) :-
        stonexy(X, Y, A, S),
        newstone(S, N),
        rev_stonexy(X, Y, N, A, B1),
        rev_h(B1, B2),
        rev_v(B2, B3),
        rev_d(X, Y, B3, B4),
        rev_stonexy(X, Y, S, B4, B).

newstone(b, bn).
newstone(w, wn).
test :-
        A = [[e,e,e,e,e,e,e,e],
             [e,e,e,e,e,e,e,e],
             [e,e,e,e,e,e,e,e],
             [e,e,e,b,w,e,e,e],
             [e,e,e,w,b,e,e,e],
             [e,e,e,e,e,e,e,e],
             [e,e,e,e,e,e,e,e],
             [e,e,e,e,e,e,e,e]],
        B = [[e,e,e,e,e,e,e,e],
             [e,e,e,e,e,e,e,e],
             [e,e,e,e,e,e,e,e],
             [e,e,e,b,w,e,e,e],
             [e,e,e,w,b,e,e,e],
             [e,e,e,b,e,e,e,e],
             [e,e,e,e,e,e,e,e],
             [e,e,e,e,e,e,e,e]],
        last_stone(A, B, X, Y),
        rev(X, Y, B, C),
        write_board(C).

rev(A, B) :-
        boarddata(C),
        last_stone(C, A, X, Y),
        rev(X, Y, A, B).

write_board([B]) :- write(B).
write_board([A|R]) :-
        write(A), write(','), nl,
        write_board(R).

clear(1).
clear :- abolish(board, 1).
invclear(1).
invclear :-
        abolish(contain, 4),
        abolish(box, 4),
        abolish(circle, 3),
        abolish(x_order, 3),
        abolish(y_order, 3),
```

```
        abolish(horizontal, 2),
        abolish(vertical, 2).
```

C.3 Entity-Relationship Diagram Editor

```
objectmap([entitymap]).
relationmap([relationshipmap]).

entitymap :-
        entity( ENTITY, ATTRIBUTES ),
        entitymap(ENTITY, ATTRIBUTES),
        fail.
entitymap(ENTITY, ATTRIBUTES) :-
        box( ENTITY, 70, 40, [green] ),
        label( [ENTITY], ENTITY, [] ),
        contain(ENTITY, [ENTITY], 0, []),
        attributemap( ENTITY, ATTRIBUTES ).

attributemap( _, [ ] ).
attributemap( ENTITY, [H | L] ) :-
        ellipse( [ENTITY | H], 50, 30, [fill] ),
        label( [[ENTITY | H]], H, [] ),
        contain( [ENTITY|H], [[ENTITY|H]], 0, [] ),
        adjacent( ENTITY, [ENTITY | H], 2 ),
        connect([ENTITY,[ENTITY|H]],ENTITY,[ENTITY|H],center,center,[straight,blue]),
        attributemap( ENTITY, L ).

relationshipmap :-
        relationship( R, ENTITY1, ENTITY2, M, N ),
        relationshipmap( R, ENTITY1, ENTITY2, M, N ),
        fail.
relationshipmap( R, ENTITY1, ENTITY2, M, N ) :-
        diamond( [R], 70, 40, [fill,green] ),
        label([[R]], R, [visible] ),
        contain([R],[[R]],0,[]),
        adjacent( ENTITY1,  ENTITY2, 4 ),
        between( [R], ENTITY1, ENTITY2, []),
        connect( [ENTITY1, [R]], ENTITY1, [R], center, center, [straight]),
        connect( [ENTITY2, [R]], ENTITY2, [R], center, center, [straight]).

invomap([ientitymap]).
invrmap([irelationshipmap]).

ientitymap :-
        ientitymap(E, A),
        asr(entity(E,A)),
        fail.
ientitymap(E, A) :-
        contain(B, L, _, _),
        box(B, _, _, _), label(L, E, _),
        iattributemap(B, E, A).
iattributemap(B, E, A) :-
        findall(L, (con(B, W), iattribute(W, L)), A), !.
iattribute(W, L) :-
        contain(W, X, _, _),
        ellipse(W, _, _, _), label(X, L, _).

irelationshipmap :-
        irelationshipmap(R, E1, E2, M, N),
        asr(relationship(R,E1,E2,M,N)),
        fail.
irelationshipmap(R, EL1, EL2, M, N) :-
        contain(D, DL, _, _),
        diamond(D, _, _, _), label(DL, R, _),
        conl( E1, D, M), conl( E2, D, N), E1 @< E2,
        contain(E1, L1, _, _), box(E1, _, _, _), label(L1, EL1, _),
        contain(E2, L2, _, _), box(E2, _, _, _), label(L2, EL2, _).

con(X, Y) :- connect(X, Y, _, _, _).
con(X, Y) :- connect(Y, X, _, _, _).
conl(X, Y, L) :- connectwithlabel(X, Y, _, _, _, L).
conl(X, Y, L) :- connectwithlabel(Y, X, _, _, _, L).
conl(X, Y, L) :-
        linedata(X, Y, Z), atom(X),
        connect(X, Y, _, _, _),
        contain(Z, LO, _, _),
```

```
        label(LO, L, _).
conl(X, Y, L) :-
        linedata(Y, X, Z), atom(X),
        connect(Y, X, _, _, _),
        contain(Z, LO, _, _),
        label(LO, L, _).

clear :-
        abolish(entity, 2),
        abolish(relationship, 5).

invclear :-
        abolish(connect, 5),
        abolish(contain, 4),
        abolish(diamond, 4),
        abolish(ellipse, 4),
        abolish(box, 4),
        abolish(label, 3),
        abolish(adjacent, 3),
        abolish(connectwithlabel, 6),
        abolish(linedata, 3),
        assert(linedata(0,0,0)).
```

C.4 Family Tree

```
% Visual mapping driver
        objectmap([personmapping]).
        relationmap([generationmapping]).

        personmapping :-
                person(X),
                personmap(X),
                fail.
        generationmapping :-
                generation( parents(F, M), children(C) ),
                generationmap( parents(F, M), children(C) ),
                fail.

% Visual mapping rules
        personmap(X) :-
                box( X, 65, 30, [fill, green] ),
                label( [X], X, [center] ),
                contain(X, [X], 0, []).
        generationmap( parents(F, M), children(C) ) :-
          % relation between father and mother
                connect2( F, M, right, left, [blue, thick, dotted] ),
                horizontal( [F, M], [rigid] ),
                x_order( [F, M], 40, [pliable] ),
          % relation between parents and children
                circle([F, M], 2, [red, bound]),
                x_average( [F, M], [F, M], [rigid] ),
                x_average( [F, M], C, [pliable] ),
                horizontal( [F, [F, M]], [rigid] ),
                ver_map( [F, M], C ),
          % relation among children
                hor_map( C ),
                fail.

% For initialize
        clear :- abolish(person, 1), abolish(generation, 2).
        invclear :-
                abolish(box, 4),
                abolish(label, 3),
                abolish(circle, 3),
                abolish(x_order, 3),
                abolish(y_order, 3),
                abolish(horizontal, 2),
                abolish(vertical, 2),
                abolish(x_average, 3),
                abolish(contain, 4),
                abolish(connect2, 5).

% Misc.
        ver_map( _, [] ).
        ver_map( P, [C | L] ) :-
                y_order( [P, C], 50, [pliable] ),
                connect2( P, C, bottom, top, [blue, orthogonal,thin, solid] ),
```

```
                ver_map( P, L ).
        hor_map( [_] ).
        hor_map( [X, Y | L] ) :-
                left_of( X, Y ), hor_map( [Y | L] ).
        left_of( X, Y ) :-
                generation( parents(X, W), _ ),
                generation( parents(H, Y), _ ),
                !,
                x_order( [W, H], 40, [pliable] ).
        left_of( X, Y ) :-
                generation( parents(X, W), _ ),
                !,
                x_order( [W, Y], 40, [pliable] ).
        left_of( X, Y ) :-
                generation( parents(H, Y), _ ),
                !,
                x_order( [X, H], 40, [pliable] ).
        left_of( X, Y ) :-
                x_order( [X, Y], 40, [pliable] ).


% Inverse visual mapping driver
        invomap([io]).
        invrmap([ir]).

        io :-   iperson(X),
                asr(person(X)),
                fail.
        ir :-   iparents(X,Y,L1,L2),
                igeneration(parents(X,Y), children(L)),
                asr(generation(parents(L1,L2),children(L))),
                fail.

% Inverse visual mapping rules
        iperson(X) :-
                contain(A, B, _, _),
                box(A, _, _, _), label(B, X, _).
        iparents(X, Y, L1, L2) :-
                connect2(X,Y,_,_,_),
                contain(X, XL, _, _),
                box(X, _, _, _), label(XL, L1, _),
                contain(Y, YL, _, _),
                box(Y, _, _, _), label(YL, L2, _).

        igeneration(parents(X,Y), children(L)) :-
                circle(Z, _, _),
                inv_horizontal([X, Z], _),
                findall(W,
                        (connect2(Z,A,_, _, _),contain(A,B,_,_),
                         box(A,_,_,_),label(B,W,_)),
                        L),
                !.
        igeneration(parents(X,Y), children([])).

% Misc.
        inv_x_order(L, G, M) :-
                x_order(K, G, M),
                part(L, K).
        inv_horizontal(L, M) :-
                horizontal(K, M),
                part(L, K).

        part(X, [A|Y]) :- head(X, [A|Y]); part(X, Y).

        head([], _).
        head([A|X], [A|Y]) :- head(X, Y).
```

Appendix D

Examples of Mapping Rules for TRIP2a

D.1 Data Structure Animation

D.1.1 Graph Structure

```
%% Abstract Objects & Relations %%
%       edge(a,b).      edge(b,c).
%       edge(a,c).      edge(a,d).
%       node(a).        node(b).
%       node(c).        node(d).

%% Mapping Rules
objectmap([o]).
relationmap([r]).

% Object Mapping
o :- node(X), nodemap(X), fail.
nodemap(X) :-
   circle(X, 20, [red]),
   label(l(X), X, [magenta]),
   contain(X,l(X),0,[]).

% Relation Mapping
r :- edge(A, B), edgemap(A, B), fail.
edgemap(A, B) :-
   connect([A,B],A,B,center,center,[blue]),
   adjacent(A, B, 2).
```

D.1.2 List Structure

```
%% Abstract Objects & Relations %%
%    cons(c1, cons(c2, cons(c3, a, nil), b),
%             cons(c4, c, nil))).

%% Mapping Rules
objectmap([]).
relationmap([v]).

% Relation Mapping
v :- data(X),!,v(X).

v(cons(P, cons( Q, A, B ), cons(R, C, D))) :-
   pstart( P ),
   v( cons( Q, A, B ) ),
   v( cons( R, C, D ) ),
   cell( P ),
   vertical( [[P, car], Q], [left_align] ),
   horizontal( [[P, cdr], R], [top_align] ),
   % vertical type composition
   y_order( [R, Q], 30, [ ] ),
   x_order( [[P, cdr], R], 30, [ ] ),
   % horizontal type composition
   arrow([P,car,a],[P, car],Q,center,topleft,[]),
   arrow([P,cdr,a],[P, cdr],R,center,lefttop,[]),
   pend.
```

```
v( cons( P, cons( Q, A, B ), nil ) ) :-
   pstart( P ),
   v( cons( Q, A, B ) ),
   cell( P ),
   diagonal( [P, cdr] ),
   verticallisting([[P,car],Q],30,[left_align]),
   arrow([P,car,a],[P,car],Q,center,topleft,[]),
   pend.

v( cons( P, A, cons( R, B, C ) ) ) :-
   pstart( P ),
   v( cons( R, B, C ) ),
   cell( P ),
   label( [P, car, l], A, [bound] ),
   contain([P, car], [P, car, l], 0, []),
   horizontallisting([[P,cdr],R],30,[top_align]),
   arrow([P,cdr,a],[P,cdr],R,center,lefttop,[]),
   pend.

v( cons( P, A, nil ) ) :-
   pstart( P ),
   cell( P ),
   label( [P, car, l], A, [bound] ),
   contain([P, car], [P, car, l], 0, []),
   diagonal( [P, cdr] ),
   pend.

cell( P ) :-
   box( [P, car], 30, 30, [] ),
   box( [P, cdr], 30, 30, [] ),
   horizontallisting([[P,car],[P,cdr]],0,[]),
   reference( P, topleft, [P, car], top ),
   reference( P, lefttop, [P, car], left ).

diagonal( X ) :-
   map( X, O ),
   connect([O,d],O,O,topright,bottomleft,[]).
```

D.2 Sorting Algorithm Animations

D.2.1 Bublesort

```
%% Abstract Objects & Relations %%
%    numlist(L) : L is a list of numbers

%% Mapping Rules
objectmap([o]).
relationmap([r]).

% Object Mapping
o :- numlist(X), barlistmap(X).
barlistmap([]).
barlistmap([H|T]) :-
        L is H * 20 + 20,
        box(l(H), 30, L, [shaded]),
        barlistmap(T).

% Relation Mapping
r :- numlist(X), listmap(X).
listmap(X) :-
        numlistmap(X, L),
        x_order(L, 10, []),
        horizontal(L, [bottom_align]).
numlistmap([],[]).
numlistmap([A|B], [X|Y]) :-
        map(l(A), X),
        numlistmap(B, Y).
```

D.2.2 Quicksort

```
%% Abstract Operations %%
%   exchange(X,Y)
%   compair(X)
%   region(X,Y,X1,Y1)
%   setpart(X)
```

```
%   end_sort
%% Abstract Objects & Relations %%
%   num(1, green).  num(2, green).
%   num(3, green).
%   numlist([1,2,3]).
%   cmp(N, low). cmp(M, hi).

%% Mapping Rules
objectmap([o,o1]).
relationmap([r,p]).
transitionmapping([t]).

% Object Mapping
o :-
        num(H, F),
        L is H * 10 + 10,
        box(l(H), 20, L, [F]),
        fail.
o1 :-
        cmp(X,L),
        box(L,20,10,[fill]),
        numlist([H|_]),
        cmpmap(X,H,L),
        y_order([l(H),L],5,[]),
        fail.
cmpmap(X,H,L) :-
        X >= 0,
        D is X * 25,
        x_relative([l(H),L],D,[]).
cmpmap(X,H,L) :-
        X < 0,
        D is X * -25,
        x_relative([L,l(H)],D,[]).

% Relation Mapping
r :- numlist(X), listmap(X).

listmap(X) :-
        numlistmap(X, L),
        x_order(L, 5, []),
        horizontal(L, [bottom_align]).

numlistmap([],[]).
numlistmap([A|B], [X|Y]) :-
        map(l(A), X),
        numlistmap(B, Y).

p :-
        numlist([H|_]),
        place(l(H),50,200,[lx,by]).

% Transition Mapping
t :-
        swap(X, Y),
        move(l(X), [clockwise]),
        move(l(Y), [clockwise]).

%% Abstract Operations
exchange(X,Y) :-
        retract(numlist(L)),
        exch(X,Y,L,L1),
        assert(numlist(L1)),
        abolish(swap, 2),
        assert(swap(X,Y)).
exch(X,X,L,L).
exch(X,Y,L,R) :-
        replace(X,'-',L,R1),
        replace(Y,X,R1,R2),
        replace('-',Y,R2,R).
replace(X,Y,L,R) :-
        replace(X,Y,[],L,R).
replace(X,Y,R,[X|L],L1) :-
        reverse(R,R1,[]),
        append(R1,[Y|L],L1).
replace(X,Y,R,[H|L],L1) :-
        replace(X,Y,[H|R],L,L1).
append([],X,X).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

```
compair(X,Y) :-
        retract(cmp(_,Y)),
        assert(cmp(X,Y)).
compair(X,Y) :-
        assert(cmp(X,Y)).

region(X,Y,P1,P2) :-
        abolish(num,2),
        numlist(L),
        setblack(L),
        getpart(X,Y,L,PL),
        setgreen(PL),
        abolish(cmp,2),
        X1 is P1 - 1, Y1 is P2 + 1,
        assert(cmp(X1,low)),
        assert(cmp(Y1,hi)).
getpart(X,Y,[X|L],PL) :-
        gethead(Y,[],[X|L],PL).
getpart(X,Y,[H|L],PL) :-
        getpart(X,Y,L,PL).
gethead(Y,R,[Y|_],PL) :-
        reverse([Y|R],PL,[]).
gethead(Y,R,[X|L],PL) :-
        gethead(Y,[X|R],L,PL).
setgreen([]).
setgreen([H|L]) :-
        retract(num(H,_)),
        assert(num(H,green)),
        setgreen(L).
reverse([],X,X).
reverse([X|L],R,T) :-
        reverse(L,R,[X|T]).
setblack([]).
setblack([H|L]) :-
        assert(num(H,black)),
        setblack(L).

setpart(X) :-
        retract(num(X,_)),
        assert(num(X,blue)).

end_sort :-
        abolish(num,2),
        numlist(L),
        setblack(L).
```

D.2.3 Mergesort

```
%% Abstract Objects & Relations %%
%   numlist1(A,B,C,D,E,F)
%   numlist2(X)
%% Abstract Operations %%
%   merged_num(X).

%% Mapping Rules
objectmap([o1, o2]).
relationmap([r1, r2]).
transitionmapping([t]).

% Object Mapping
o1 :-
   numlist1(A,B,C,D,E,F),
   barobjlistmap(A),
   bluebarobjlistmap(B),
   nullbarobjlistmap(C),
   greenbarobjlistmap(D),
   nullbarobjlistmap(E),
   barobjlistmap(F).
o2 :-
   numlist2(X), redbarobjlistmap(X).

barobjlistmap([]).
barobjlistmap([H|T]) :-
   L is H * 10 + 10,
   box(l(H), 20, L, [shaded]),
```

```
   barobjlistmap(T).

redbarobjlistmap([]).
redbarobjlistmap([H|T]) :-
   L is H * 10 + 10,
   box(l(H), 20, L, [shaded]),
   redbarobjlistmap(T).

bluebarobjlistmap([]).
bluebarobjlistmap([H|T]) :-
   L is H * 10 + 10,
   box(l(H), 20, L, [red]),
   bluebarobjlistmap(T).

greenbarobjlistmap([]).
greenbarobjlistmap([H|T]) :-
   L is H * 10 + 10,
   box(l(H), 20, L, [green]),
   greenbarobjlistmap(T).

nullbarobjlistmap([]).
nullbarobjlistmap([H|T]) :-
   box(c(H), 20, 20, [invisible]),
   nullbarobjlistmap(T).

% Transition Mapping
t :-    merged_num(X),
        move(l(X),[up,right]).

% Relation Mapping
r1 :-
   numlist1(A,B,C,D,E,F),
   barlistmap(A, A1),
   barlistmap(B, B1),
   nullbarlistmap(C, C1),
   barlistmap(D, D1),
   nullbarlistmap(E, E1),
   barlistmap(F, F1),
   connectlist([A1,B1,C1,D1,E1,F1], L),
   x_order(L, 5, []),
   horizontal(L, [bottom_align]),
   L = [H|_],
   place(H,100,200,[lx,by]).

r2 :-
   numlist1(_,_,_,D,E,_),
   numlist2(L),
   barlistmap(L, LM),
   x_order(LM, 5, []),
   horizontal(LM, [bottom_align]),
   tail(L, L1), tail2(E, D, CE1),
   map(l(L1), LL1),
   y_relative([CE1, LL1], 150, [bottom_align]),
   vertical([CE1, LL1], []).

barlistmap([],[]).
barlistmap([A|B], [X|Y]) :-
   map(l(A), X),
   barlistmap(B, Y).
nullbarlistmap([],[]).
nullbarlistmap([A|B], [X|Y]) :-
   map(c(A), X),
   nullbarlistmap(B, Y).
connectlist([A], A).
connectlist([A|B], L) :-
   connectlist(B, L1),
   append(A, L1, L).

tail([A], A).
tail([A|B], L) :- tail(B, L).

tail2([], D, CE1) :-
        tail(D, D1), map(l(D1), CE1).
tail2(E, _, CE1) :-
        tail(E, E1), map(c(E1), CE1).
```

D.2.4 Heapsort

```
%% Abstract Objects & Relations %%
%       tree(a, [b,c,d]).
%       tree(b, [e,f]).
%       node(a).       node(b).
%       node(c).       node(d).
%       node(e).       node(f).

%% Mapping Rules
objectmap([nodemap]).
relationmap([relmap,relmap1]).

% Object Mapping
nodemap :- node(X, M), nodemap(X, M), fail.

nodemap(X, M) :-
   circle(c(X), 20, [M, purple]),
   label([c(X)], X, [center]),
   contain(c(X), [c(X)], 0, []).

% Relation Mapping
relmap  :- tree(X, Y), treemap(X, Y), fail.
treemap(A, [H|L]) :-
   id_map([H|L], [CH|CL]),
   horizontal([CH|CL], []),
   x_average(c(A), [CH|CL], []),
   y_order([c(A),CH],20,[]),
   mconnect(c(A),[CH|CL],center,center,[blue]).

relmap1 :- adjacent(X, Y), adjmap(X, Y), fail.
adjmap(P,Q) :-
   x_order([c(P),c(Q)],20, []).

% misc.
adjacent(X, Y) :-
   tree(A, B), first2(B, L, R),
   leftmost(L, X), rightmost(R, Y).
first2([L, R|_], L, R).
first2([_|A], L, R) :- first2(A, L, R).
leftmost(A, X) :- tree(A, L),
                  tail(L, B), !,
                  leftmost(B,X).
leftmost(A, A).
rightmost(A, X) :- tree(A, [H|L]), !,
                   rightmost(H, X).
rightmost(A, A).
tail([X], X).
tail([H|L], X) :- tail(L, X).
mconnect(_, [], _, _, _).
mconnect(A, [H|L], L1, L2, MODE) :-
   connect([A|H], A, H, L1, L2, MODE),
   mconnect(A, L, L1, L2, MODE).
id_map([], []).
id_map([H|L], [c(H)|CL]) :- id_map(L, CL).
```

D.3 The Tower of Hanoi

```
%% Abstract Objects & Relations %%
%       towers(at_x, at_y, at_z)
%% Abstract Operations %%
%       hmove(X, Y) : move from X to Y

%% Mapping Rules
objectmap([o]).
relationmap([r]).
transitionmapping([t]).

% Object Mapping
o :-
        box(a, 20, 10, [shaded]),
        box(b, 40, 10, [shaded]),
        box(c, 60, 10, [shaded]),
        box(d, 80, 10, [shaded]),
        box(e,100, 10, [shaded]),
```

```
        box(x,120, 10, [fill]),
        box(y,120, 10, [fill]),
        box(z,120, 10, [fill]),
        place(x, 100, 100, []).
% Relation Mapping
r :-
        horizontallisting([x,y,z],0,[]),
        towers(A, B, C),
        verticallisting(A,5,[]),
        verticallisting(B,5,[]),
        verticallisting(C,5,[]).
% Transition Mapping
t :-
        go(X), move(X, [up]).

%% Abstract Operations

hmove(X, Y) :- move(_, X, Y).
move(N, x, y) :-
        towers([H|A], B, C),
        abolish(go, 1),
        assert(go(H)),
        abolish(towers, 3),
        assert(towers(A, [H|B], C)).
move(N, x, z) :-
        towers([H|A], B, C),
        abolish(go, 1),
        assert(go(H)),
        abolish(towers, 3),
        assert(towers(A, B, [H|C])).
move(N, y, x) :-
        towers(A, [H|B], C),
        abolish(go, 1),
        assert(go(H)),
        abolish(towers, 3),
        assert(towers([H|A], B, C)).
move(N, y, z) :-
        towers(A, [H|B], C),
        abolish(go, 1),
        assert(go(H)),
        abolish(towers, 3),
        assert(towers(A, B, [H|C])).
move(N, z, x) :-
        towers(A, B, [H|C]),
        abolish(go, 1),
        assert(go(H)),
        abolish(towers, 3),
        assert(towers([H|A], B, C)).
move(N, z, y) :-
        towers(A, B, [H|C]),
        abolish(go, 1),
        assert(go(H)),
        abolish(towers, 3),
        assert(towers(A, [H|B], C)).
```

D.4 Bin Packing Animation

```
%% Abstract Operation %%
%    try(ID, VALUE).
%    put_at(ID, NUM).
%% Abstract Objects & Relations %%
%    bin(ID,V).
%    binlist([[ID1,ID2,..],[ID3,ID4,..],..]).

%% Mapping Rules
objectmap([fr_map,bin_objmap]).
relationmap([layout_map]).
transitionmapping([trans_map]).

% Object Mapping
fr_map :-
   box(frame,400,200,[blue,bound]),
   place(frame,300,200,[]).
bin_objmap :-
   bin(ID, V),
   HF is V * 200, floor(HF, HI),
```

```
   box(ID,30,HI,[shaded]),
   fail.

% Transition Mapping
trans_map :-
   mv(X), move(X,[up]).

% Relation Mapping
layout_map :-
   binlist(L),
   L = [H|T],
   try_bin_map(H),
   bin_list_vertical_map(T),
   bin_list_horizontal_map(T).

bin_list_vertical_map([]).
bin_list_vertical_map([H|T]) :-
   verticallisting(H,0,[]),
   bin_list_vertical_map(T).

bin_list_horizontal_map([]).
bin_list_horizontal_map(L) :-
   taillist(L, [H|TL]),
   contain(frame,H,0,
           [left_align,bottom_align]),
   horizontallisting([H|TL],0,
                     [bottom_align]).

try_bin_map([]).
try_bin_map([T]) :-
   horizontallisting([T,frame],50,
                     [bottom_align]).

%% Abstract Operations

try(ID, V) :-
        assert(bin(ID, V)),
        retract(binlist([H|L])),
        assert(binlist([[ID]|L])).

put_at(ID, N) :-
        abolish(mv,1),
        assert(mv(ID)),
        retract(binlist(L)),
        put_at(ID, N, L, []).
put_at(ID, N, [], [[A|B]|P]) :-
        N =:= 0,
        reverse([B|P], RP),
        append(RP,[[ID]],RES),
        assert(binlist(RES)).
put_at(ID, N, [H|L], [[A|B]|P]) :-
        N =:= 0,
        reverse([B|P], RP),
        append(RP,[[ID|H]|L],RES),
        assert(binlist(RES)).
put_at(ID, N, [H|F], P) :-
        N1 is N - 1,
        put_at(ID, N1, F, [H|P]).

% misc.

taillist([H],[HT]) :- tail(H,HT).
taillist([H|R],[HT|HR]) :-
   tail(H,HT), taillist(R,HR).

tail([T],T).
tail([_|R],T) :- tail(R, T).
```

D.5 Finding a Minimum Spanning Tree

```
%% Abstract Objects & Relations %%
%   set([a,b,c]).
%   set([d]).
%   set([e]).
%   edges([e(..),...,e(..)],[....],[...]).
```

```
%% Mapping Rules
objectmap([o,o1]).
relationmap([r1,r2,r3,r4]).

% Object Mapping
o :-
        set(X),
        nodemap(X),
        fail.
nodemap([]).
nodemap([X|Y]) :-
        circle(X, 15, [blue]),
        label([X],X,[]),
        contain(X,[X],0,[]),
        nodemap(Y).
o1 :-
        place(c,150,300,[]).

% Relation Mapping
r1 :-   % for connecting edge of graph
        edges(L1,L2,L3),
        edges_objmap1(L1).
edges_objmap1([]).
edges_objmap1([H|L]) :-
        edgemap(H),
        edges_objmap1(L).
edgemap(e(X,Y,V,in)) :-
   connectwithlabel([X|Y],X,Y,center,center,
                              [blue,thick],V).
edgemap(e(X,Y,V,out)) :-
   connectwithlabel([X|Y],X,Y,center,center,
                              [black,thin],V).
edgemap(e(X,Y,V,yet)) :-
   connectwithlabel([X|Y],X,Y,center,center,
                              [blue,thick],V).

r2 :-   % for yet-inserted edges
        edges(L1,L2,L3),
        append(L2,L3,L),
        edges_objmap2(L).
edges_objmap2([]).
edges_objmap2([H|L]) :-
   circle([H|1],5,[]),
   circle([H|2],5,[]),
   horizontallisting([[H|1],[H|2]],50,[]),
   H = e(X,Y,V,_),
   connectwithlabel([X|Y],[H|1],[H|2],
              center,center,[blue,thick],V),
   connectwithlabel([X,Y],X,Y,
              center,center,[black,thin],V),
   edges_objmap2(L).

r3 :-
        edges(L1,L2,L3),
        edges_relmap(L2,L3).
edges_relmap([],[]).
edges_relmap([],L) :-
        create_olist(L, OL),
        tail(OL,T),
        place(T,350,150,[]),
        verticallisting(OL, 25, []).
edges_relmap(L,[]) :-
        create_olist(L, OL),
        tail(OL,T),
        place(T,350,165,[]),
        verticallisting(OL, 25, []).
edges_relmap(L1,L2) :-
        create_olist(L1, OL1),
        create_olist(L2, OL2),
        tail(OL2,T2), place(T2,350,150,[]),
        tail(OL1,T1), OL2 = [H2|_],
        verticallisting([T1,H2], 40, []),
        verticallisting(OL1, 25, []),
        verticallisting(OL2, 25, []).
create_olist([],[]).
create_olist([H|L],[[H|1]|OL]) :-
```

```
        create_olist(L, OL).
tail([T],T).
tail([H|L],T) :- tail(L, T).

r4 :-
        adj_edge(L),
        adj_edgemap(L).
adj_edgemap([]).
adj_edgemap([E|L]) :-
        E = e(X,Y,_,_),
        adjacent(X,Y,2),
        adj_edgemap(L).
```

Bibliography

[1] A. Frick and H. Mehldau and A. Ludwig. A fast adaptive layout algorithm for undirected graphs. In *International Workshop on Graph Drawing '94 (Proc. GD'94) (LNCS 894)*, pages 388–403, 1994.

[2] Christopher Ahlberg and Ben Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Human Factors in Computing Systems. Conference Proceedings CHI'94*, pages 313–317, 1994.

[3] Apple Computer, Inc. *Inside Macintosh*, 1985.

[4] Danielle Argiro, Mark Young, Steve Kubica, and Steve Jorgensen. *Enabling Technologies for Computational Science: Frameworks, Middleware and Environments*, chapter Chapter 12: Khoros – An Integrated Development Environment for Scientific Computing and Visualization, pages 147–157. Kluwer Academic Publishers, March 2000.

[5] Akihiro Baba and Jiro Tanaka. Eviss: A visual system having a spatial parser generator. In *Proceedings of Asia Pacific Computer Human Interaction 1998 (APCHI'98)*, pages 158–164. IEEE Computer Society Press, July 1998.

[6] R.M. Baecker. Sorting our sorting. Distributed by Morgan Kaufmann, Publishers, 1981. 30-minute color sound videotape.

[7] L. Bartram, A. Ho, J. Dill, and F. Henigman. The continuos zoom: A constrained fisheye technique for viewing and navigating large information spaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'95)*, pages 207–215, 1995.

[8] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis Tollis. Annotated bibliography on graph drawing algorithms. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.

[9] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation: Tutorial and user manual. Computing Science Technical Report 132, AT&T Bell Laboratories, January 1987.

[10] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation. In *Computing Systems*, volume 4, pages 5–30. USENIX, winter 1991.

[11] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[12] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *ACM Object-Oriented Programming Systems, Languages, and Applications*, pages 48–60, 1987.

[13] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, volume 7, pages 87–96, 1997.

[14] Brad A. Myers and Richard G. McDaniel and Robert C. Miller and Alan S. Ferrency and Andrew Faulring and Bruce D. Kyle and Andrew Mickish and Alex Klimovitski and Patrick Doane. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[15] Marc H. Brown. *Algorithm Animation*. MIT Press, Massachusetts, 1988.

[16] Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Proceedings 1991 IEEE Workshop on Visual Languages*, pages 4–9, October 1991.

[17] Marc H. Brown and John Hershberger. Color and sound in algorithm animation. *IEEE Computer*, pages 52–63, December 1992.

[18] Bruce H. Thomas and Paul Calder. Animating direct manipulation interfaces. In *UIST'95*, pages 3–12, 1995.

[19] Stuart K. Card, George G. Robertson, and Jack D. Mackinlay. The information visualizer, an informaiton workspace. In *ACM Human Factors in Computing Systems*, pages 181–188, 1991.

[20] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 45–56, November 1993.

[21] S. K. Chang. *Principles of Visual Programming Systems*. Prentice-Hall, New Jersey, 1990.

[22] Shi-Kuo Chang. Picture processing grammar and its applications. *Information Sciences*, 3:121–148, 1971.

[23] Takashi Chikayama. KLIC: A Portable and Parallel Implementation of a Concurrent Logic Programming Language. In Takayasu Ito and Robert H. Halstead Jr., editor, *Proceedings of International Workshop PSLS '95 (Volume 1068 of Lecture Notes in Computer Science)*, pages 286–294. Springer-Verlag, October 1995.

[24] Chuck Clanton, Jock Mackinlay, Dave Ungar, and Emilie Young. Animation of user interfaces, November 1992. Panel at UIST'92.

[25] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[26] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[27] Aldus corporation. Intellidraw, 1992. (Commercial software).

[28] Kenneth C. Cox and Gruia-Catalin Roman. Visualizing concurrent computations. In *Proceedings 1991 IEEE Workshop on Visual Languages*, pages 18–24, 1991.

[29] Philip T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *Proc. IEEE Work. Visual Languages, VL*, pages 150–156, Los Alamitos, California, 4–6 1989. IEEE CS Press.

[30] Craig Upson and Thomas Faulhaber, Jr. and David Kamins and David Laidlaw and David Schlegel and Jeffrey Vroom and Robert Gurwitz and Andries van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics & Applications*, 9(4):30–42, July 1989.

[31] Allen Cypher. Eager : Programming repetitive tasks by example. In *ACM Human Factors in Computing Systems*, pages 33–39, 1991.

[32] Ian H. Witten David L. Maulsby and Kenneth A. Kittlitz. Metamouse: Specifying graphical procedures by example. *Computer Graphics*, 23(3):127–136, July 1989.

[33] Luiz DeRose and Daniel A. Reed. SvPablo: A Multi-Language Architecture-IndependentPerformance Analysis System. In *Proceedings of the International Conference on Parallel Processing (ICPP'99)*, pages 311–318, September 1999.

[34] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*. AMAST Series. World Scientific, 1998.

[35] Duane Hanselman and Bruce R. Littlefield. *Mastering MATLAB 6*. Prentice Hall, 2000.

[36] Rovert Adamy Duisberg. Animation using temporal constraints: An overview of the animus system. *Human-Computer Interaction*, 3(3):275–307, 1987/88.

[37] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, pages 149–160, 1984.

[38] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[39] Nobuo Saito et al. Chapter 3: Research on the COS operating system for massively parallel system. In *6th Symposium on Fundamental System of Information Computation based on Super Parallel Principle (In Japanese)*, pages (3–1) – (3–64), March 1995.

[40] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice, Second Edition*. Addison-Wesley, 1990.

[41] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, Jan 1990.

[42] G. L. Fisher and D. E. Busse and D. A. Wolber. Adding rule-based reasoning to a demonstrational interface builder. In *Proceedings of ACM User Interface Software and Tecnology (UIST'92)*, pages 89–97. ACM Press, 1992.

[43] George W. Furnas. Generalized fisheye views. In *ACM Human Factors in Computing Systems*, pages 16–23, April 1986.

[44] Eric J. Golin. Interaction Diagrams: a visual language for controlling a visual program editor. In *Proceedings 1991 IEEE Workshop on Visual Languages*, pages 153–158, 1991.

[45] Eric J. Golin and Steven Reiss. The specification of visual language syntax. In *IEEE Workshop on Visual Languages*, pages 105–110, 1989.

[46] Ronny Hadany and David Harel. A multi-scale method for drawing graphs nicely. In *Proceedings of 25th International Workshop on Graph-Theoretic Concepts in Computer Scienece (LNCS 1665)*, pages 262–277, 1999.

[47] David Harel and Yehuda Koren. A fast multi-scale method for drawing large graphs. In *8th International Symposium, GD2000 (LNCS 1984)*, pages 183–196, 2000.

[48] H. Rex Hartson and Deborah Hix. Human-computer interface development: Concepts and systems for its management. *ACM Computing Surveys*, 21(1):5–92, March 1989.

[49] Tyson Henry and Scott Hudson. Interactive graph layout. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 55–64, November 1991.

[50] Michael Himsolt. Graphed: A graphical platform for the implementation of graph algorithms (extended abstract and demo). In *International Workshop on Graph Drawing '94 (Proc. GD'94) (LNCS 894)*, pages 182–193, 1994.

[51] Hiroshi Hosobe. A scalable linear constraint solver for user interface construction. *Proc. 6th Int'l Conf. on Principles and Practice of Constraint Programming (CP2000)*, pages 218–232, Sep. 2000. Lecture Notes in Computer Science.

[52] Hiroshi Hosobe. A Hierarchical Framework for Integrating Constraints with Graph Layouts. In *Human Computer Interaction–IFIP INTERACT 2001*, pages 704–705, 2001. (short paper).

[53] Hiroshi Hosobe. A Modular Geometric Constraint Solver for User Interface Applications. In *Proc. of ACM Symp. on User Interface Software and Technology (UIST'2001)*, pages 91–100, 2001.

[54] Hiroshi Hosobe and Shinichi Honiden. A Constraint-Based Approach to Information Visualization for XML. In *Interaction 2001*, number 5 in IPSJ Symposium Series Vol.2001, pages 83–90, March 2001. (In Japanese).

[55] Hiroshi Hosobe, Ken Miyashita, Shin Takahashi, Satoshi Matsuoka, and Akinori Yonezawa. Locally simultaneous constraint satisfaction. In *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming*, No.874 in Lecture Notes in Computer Science, pages 51–62. Springer-Verlag, 1994.

[56] Takeo Igarashi, Satoshi Matsuoka, Sachiko Kawachiya, and Hidehiko Tanaka. Interactive beautification: A technique for rapid geometric design. In *ACM Annual Symposium on User Interface Software and Technology (UIST'97)*, pages 105–114, 1997.

[57] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3d freeform design. In *SIGGRAPH 99 Conference Proceedings*, pages 409–416, August 1999.

[58] J. Rekers and A. Schurr. A graph grammar approach to graphical parsing. In *Proc. VL'95 11th International IEEE Symposium on Visual Languages*, pages 195–202, September 1995.

[59] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Fourteenth ACM Symposium on the Principle of Programming Languages*, pages 111–119, 1987.

[60] James A. Landay and Brad A. Myers. Interactive sketching for the early stages of user interface design. In *Proceedings of CHI '95: Human Factors in Computing Systems*, pages 43–50, May 1995.

[61] John T. Stasko and Joseph F. Wehrli. Three-dimensional computation visualization. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 100–107, 1993.

[62] Sucktae Joung and Jiro Tanaka. Generating a visual system with soft layout constraints. In *Proceedings of the International Conference on Information (Information'2000)*, pages 138–145, 2000.

[63] Kenneth M. Kahn. Concurrent constraint programs to parse and animate pictures of concurrent constraint programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 943–950. ICOT, 1992.

[64] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 7–15, October 1990.

[65] T. Kamada. *Visualizing Abstract Objects and Relations, A Constraint-Based Approach*. World Scientific, Singapore, 1989.

[66] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, April 1989.

[67] Tomihisa Kamada and Satoru Kawai. A General Framework for Visualizing Abstract Objects and Relations. *ACM Transactions on Graphics*, 10(1):1–39, Jan. 1991.

[68] Kathy Ryall and Joe Marks and Stuart M. Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of UIST'97*, pages 97–104, October 1997.

[69] Kim Marriott and Peter Moulder and Peter J. Stuckey and Alan Borning. Solving disjunctive constraints for interactive graphical applications. In *Principles and Practive of Constraint Programming – CP2001*, pages 361–376, November 2001.

[70] D. E. Knuth. Computer-drawn flowcharts. *Communications of the ACM*, 6(9):555–563, September 1963.

[71] Hideki Koike. Fractal views: A fractal-based method for controlling information display. *ACM Transactions on Information Systems*, 13(3):305–323, 1995.

[72] David Kurlander and Seven Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics*, 12(4):277–304, October 1993.

[73] Fred H. Lakin. Computing with text-graphic forms. In *Proceedings of the 1980 ACM Conference on Lisp and functional programming*, pages 100–106, 1980.

[74] Fred H. Lakin. A structure from manipulation for text-graphic objects. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques (SIGGRAPH'80)*, pages 100–107, July 1980.

[75] Fred H. Lakin. Visual grammars for visual languages. In *AAAI-87: Sixth National Conference on Artificial Intelligence*, pages 683–688, July 1987.

[76] John Lasseter. Principles of traditional animation applied to 3D computer animation. *ACM Computer Graphics*, 21(4):35–44, July 1987.

[77] John Levine, Tony Mason, and Doug Brown. *lex & yacc, 2nd Edition*. O'Reilly& Associates, 1992.

[78] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[79] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, April 1986.

[80] John Maloney, Alan Borning, and Bjorn N. Freeman-Benson. Constraint technology for user-interface in ThingLabII. In *OOPSLA '89 Conference Proceedings*, pages 381–388, 1989.

[81] Mark D. Gross and Ellen Do. Ambiguous intentions: A paper-like interface for creative design. In *Proceedings ACM Conference on User Interface Software Technology (UIST) '96*, pages 183–192, 1996.

[82] Kim Marriott. Constraint multiset grammars. In *Proceedings IEEE Symposium on Visual Languages*, pages 118–125. IEEE Society Press, 1994.

[83] Toshiyuki Masui. Evolutionary learning of graph layout constraints from examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, volume 7, pages 103–108, 1994.

[84] The MathWorks Inc. *MATLAB 6.5*. http://www.mathworks.com/products/matlab/.

[85] Satoshi Matsuoka, Shin Takahashi, Tomihisa Kamada, and Akinori Yonezawa. A General Framework for Bi-directional Translation between Abstract and Pictorial Data. *ACM Transactions on Information Systems*, 10(4):408–437, October 1992.

[86] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, 1995.

[87] Ken Miyashita, Satoshi Matsuoka, Shin Takahashi, and Akinori Yonezawa. Interactive generation of graphical user interfaces by multiple visual examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 85–94, 1994.

[88] Ken Miyashita, Satoshi Matsuoka, Shin Takahashi, Akinori Yonezawa, and Tomihisa Kamada. Declarative Programming of Graphical Interfaces by Visual Examples. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'92)*, pages 107–116, November 1992.

[89] Brad A. Myers. Amulet project home page. http://www.cs.cmu.edu/∼amulet.

[90] Brad A. Myers and William Buxton. Creating highly-interactive and graphical user interfaces by demonstration. *Computer Graphics : Proceedings of the 13th annual conference on computer graphics and interactive techniques*, 20(4):249–258, 1986.

[91] Brad A. Myers et al. Garnet, comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[92] Ken Nakayama, Satoshi Matsuoka, and Satoru Kawai. Visualization of abstract concepts using generalized path binding. In *Proceedings of CG International '90*, pages 377–402. Springer-Verlag, 1990.

[93] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN*, pages 12–26, August 1973.

[94] G. Nelson. Juno, a constraint-based graphics system. *Computer Graphics*, 19(3):235–243, July 1985.

[95] NeXT Inc. *NeXT System Reference Manual*.

[96] Tohru Ogawa and Jiro Tanaka. Visualization of program execution via customized view. In *Proceedings of 5th Asia Pacific Conference on Computer Human Interaction (APCHI2002)*, volume 2, pages 823–832, November 2002.

[97] Daisuke Okajima and Masami Hagiya. Inverse Transformation of XSLT. In *SPA2000 online proceedings*. Japan Society for Software Science and Technology (JSSST), http://www.jaist.ac.jp/SPA2000/proceedings/, 2000. (In Japanese).

[98] Havoc Pennington. *GTK+/GNOME Application Development*. Que, 1999.

[99] L. A. Pineda et al. GRAFLOG: Programming with interactive graphics and prolog. In *Proceedings of CG International'88*, pages 469–478. Springer-Verlag, 1988.

[100] Daniel A. Reed, Ruth. A. Aydt, Roger J. Noe, Philip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113, October 1993.

[101] J. Rekers and Andy Schurr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[102] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone Trees: Animated 3D Visualizations of Hierarchical Information. In *ACM Human Factors in Computing Systems*, pages 189–194. SIGCHI, April/May 1991.

[103] G. C. Roman and K. C. Cox. A declarative approach to visualizing concurrent commputations. *IEEE Computer*, 22(10):25–36, October 1989.

[104] S. F. Roth et al. Interactive graphic design using automatic presentation knowledge. In *Proceedings CHI'94 Human Factors in Computing Systems*, pages 112–117. ACM Press, 1994.

[105] Nobuhiro Sakai. Visualization of abstract data. Master's thesis, Tokyo Institute of Technology, 1990. (In Japanese).

[106] Georg Sander and Adrian Vasiliu. ILOG JViews Graph Layout: a Java library for highly demanding graph-based applications. In *Electronic Notes in Theoretical Computer Science*, volume 72, No.2, 2002. First International Conference on Graph Transformation, Graph-Based Tools (GraBaTs 2002).

[107] Scott E. Hudson and Ian Smith. Ultra-lightweight constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'96)*, pages 147–155, 1996.

[108] Scott E. Hudson and John T. Stasko. Animation support in a user interface toolkit: Flexible, robust, and reusable abstractions. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 57–68, November 1993.

[109] B. Shneiderman. Direct Manipulation: A step beyond programming languages. *IEEE Computer*, pages 57–69, August 1983.

[110] N. C. Shu. *Visual Programming*. Van Nostrand Reinhold, 1988.

[111] SICS. *SICStus Prolog*. http://www.sics.se/sicstus/.

[112] Tom Sawyer Software. http://www.tomsawyer.com.

[113] Stanford University. *InterViews Reference Manual*.

[114] John T. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.

[115] John T. Stasko. Simplifying algorithm animation with tango. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 1–6, October 1990.

[116] John T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.

[117] John T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *ACM Human Factors in Computing Systems, CHI'91 Conference Proceedings*, pages 307–314, 1991.

[118] John T. Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning? — an empirical study and analysis. In *Human Factors in Computing Systems — INTERCHI'93 Conference Proceedings*, pages 61–66, 1993.

[119] Shin Takahashi, Satoshi Matsuoka, Ken Miyashita, Hiroshi Hosobe, and Tomihisa Kamada. A constraint-based approach for visualization and animation. *Constraints: An International Journal*, 3(1):61–86, 1998.

[120] Shin Takahashi, Satoshi Matsuoka, Akinori Yonezawa, and Tomihisa Kamada. A General Framework for Bidirectional Translation between Abstract and Pictorial Data. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, volume 4, pages 165–174, November 1991.

[121] Shin Takahashi, Ken Miyashita, Satoshi Matsuoka, and Akinori Yonezawa. A framework for constructing animations via declarative mapping rules. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, volume 10, pages 314–322, 1994.

[122] Kenjiro Taura and Akinori Yonezawa. Schematic: A concurrent object-oriented extension to scheme. In *Proceedings of Workshop on Object-Based Parallel and Distributed Computation (Volume 1107 of Lecture Notes in Computer Science)*, pages 59–82. Springer-Verlag, 1996.

[123] W. Teitelman and L. Masinter. The interlisp programming environment. *IEEE Computer*, 14(4):25–34, April 1981.

[124] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 158–167, 1989.

[125] Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley Pub. Co., 1999.

[126] Josie Wernecke. *The Inventor Mentor*. Addison Wesley, 1994.

[127] Kent Wittenburg and Louis Weitzman. Unification-based grammars and tabular parsing for graphical languages. *Journal of Visual Languages and Computing*, 2(4):347–370, 1991.

[128] David Wolber. Pavlov: Programming by stimulus-response demonstration. In *Human Factors in Computing Systems : CHI'96 Conference Proceedings*, pages 252–259. ACM Press, 1996.

[129] Stephen Wolfram. *Mathematica, A System for Doing Mathematics by Computer*. Addison-Wesley, 1988.

[130] H. K. T. Wong and I. Kuo. GUIDE: A Graphical User Interface for Database Exploration. In *Proceedings of the Conference on Very Large Databases*, pages 22–32, 1982.

[131] Bradley T. Vander Zanden. *Incremental Constraint Satisfaction And Its Application To Graphical Interfaces*. PhD thesis, Department of Computer Science, Cornell University, October 1988.

Index

abstract operation, 38
AR, 32
ASR, 32

instant actions, 87

long actions, 87

PR, 34

transition mapping rules, 38
transition operation, 38

visual (and inverse visual) mapping rule set, 32
visual mapping, 35
VSR, 32