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.

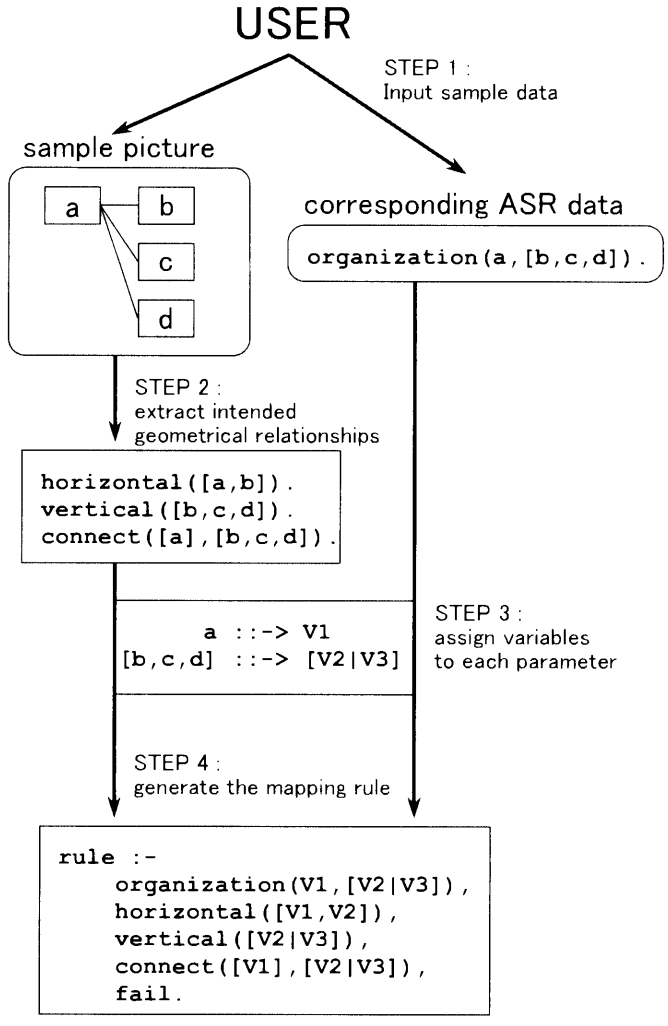


Figure A.1: Programming by visual example — a process of generating visual mapping rules.

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.

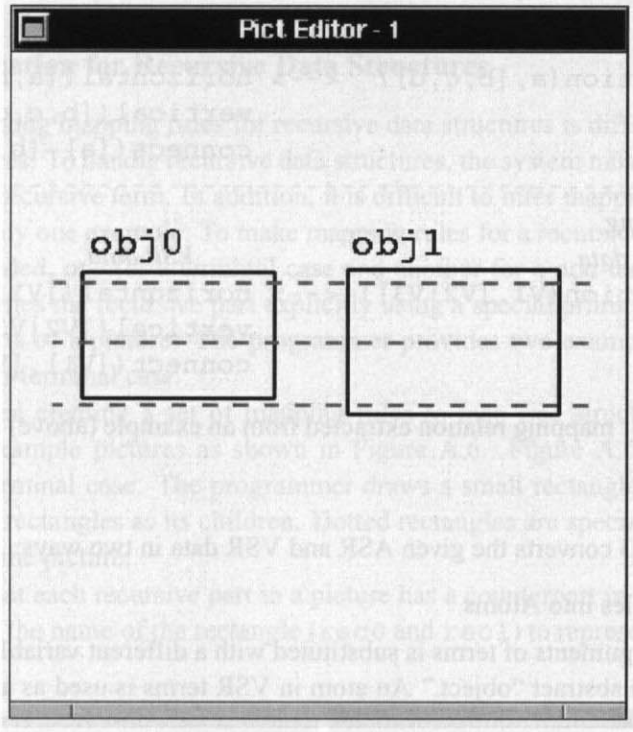


Figure A.2: TRIP3 — picture editor.

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.

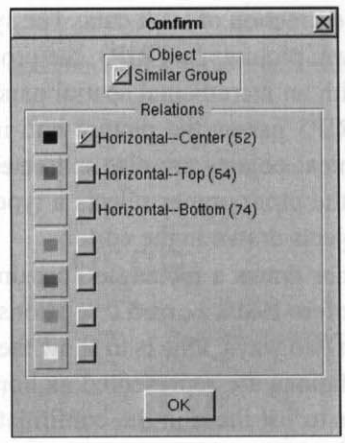


Figure A.3: TRIP3 — confirmation panel.

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.

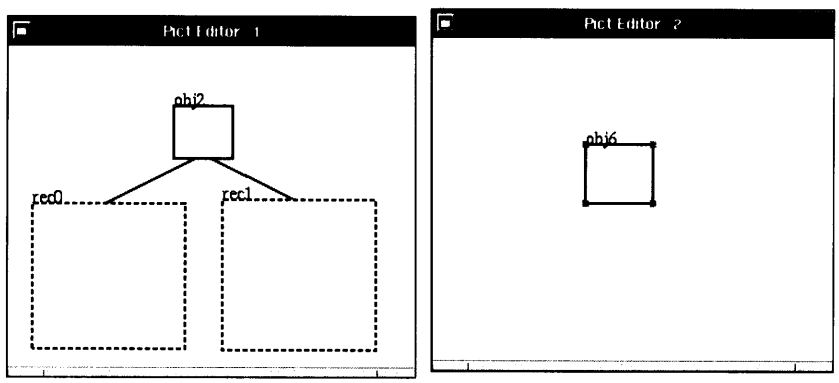
of the template, and the generalized VSR terms are put into the lower half. The generated rule is a Prolog clause, which is used for mapping between ASR data and VSR data. Figure A.1 shows the resulting mapping rule.

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.

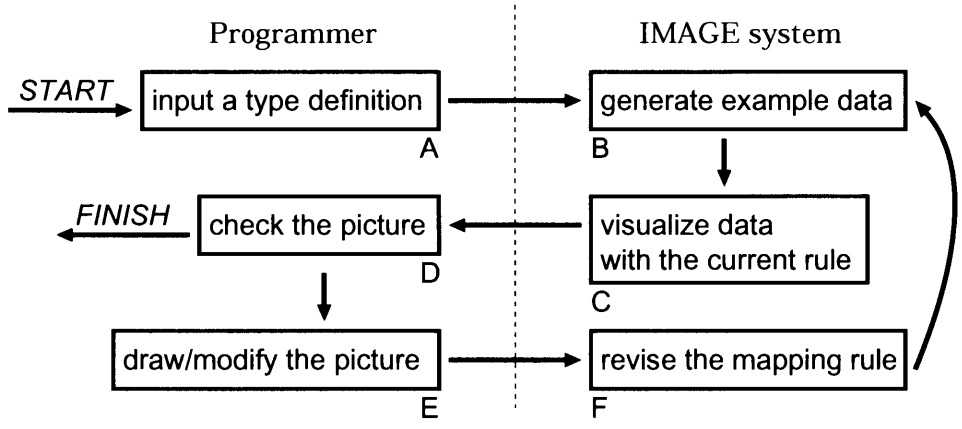


Figure A.9: Interaction between a programmer and IMAGE.

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:

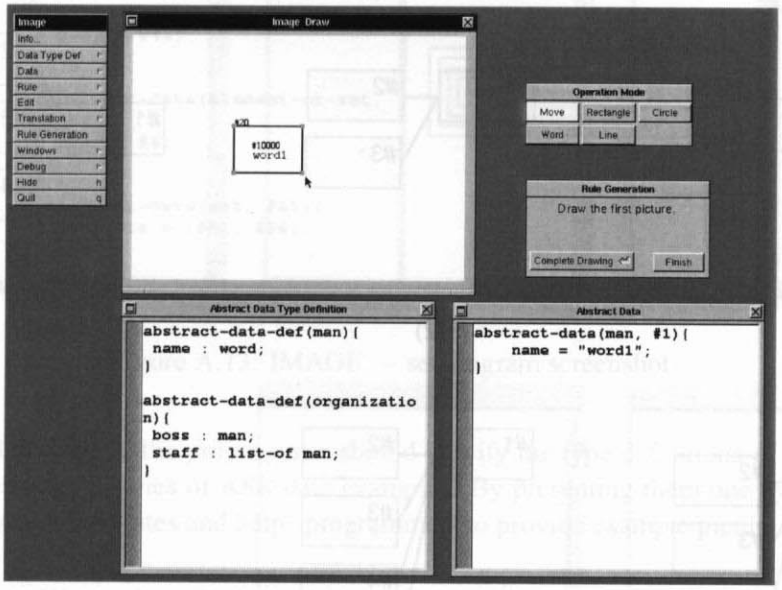


Figure A.10: Screenshot of the IMAGE system.

```
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 `mans` 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)).

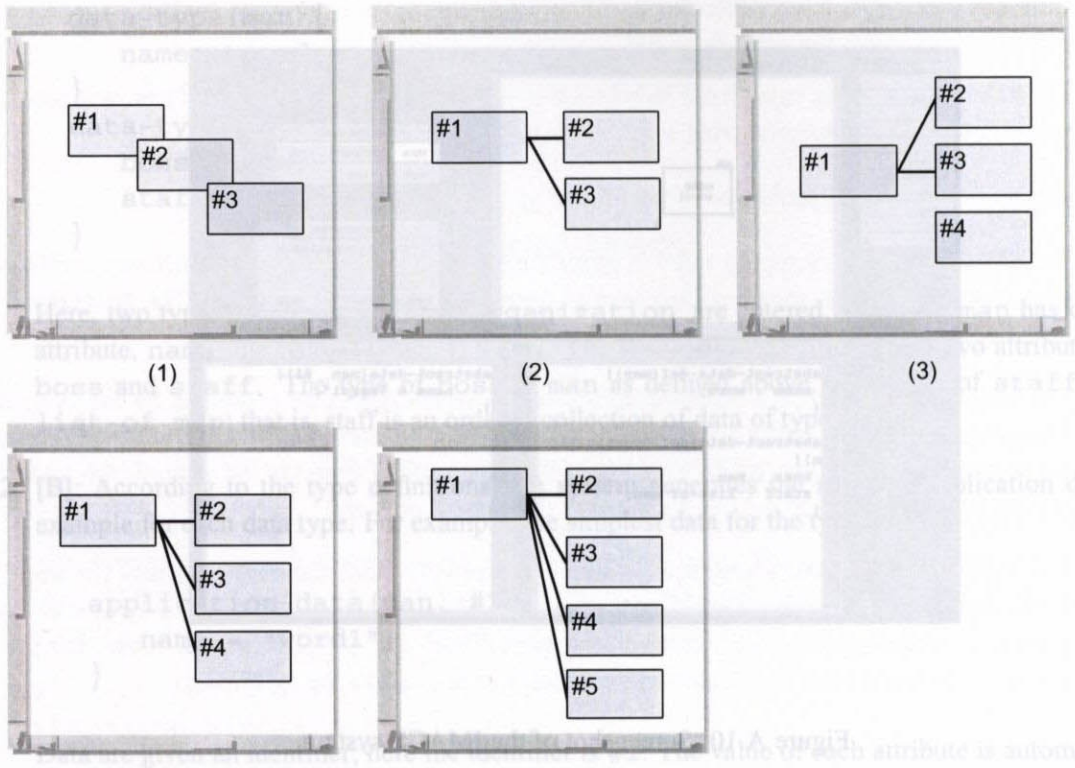


Figure A.11: IMAGE — screenshots of drawing editor in rule generation for "organization".

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.

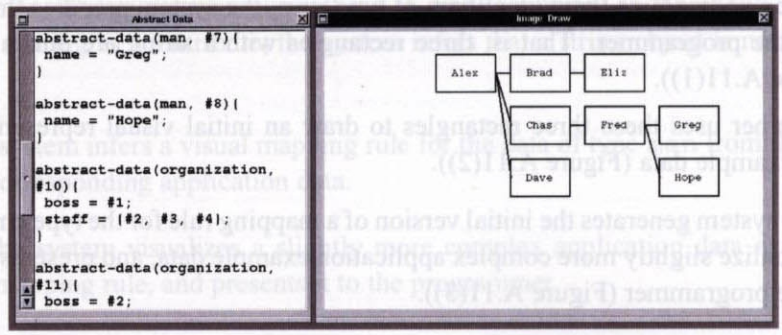


Figure A.12: IMAGE — organization diagram screenshot.

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.

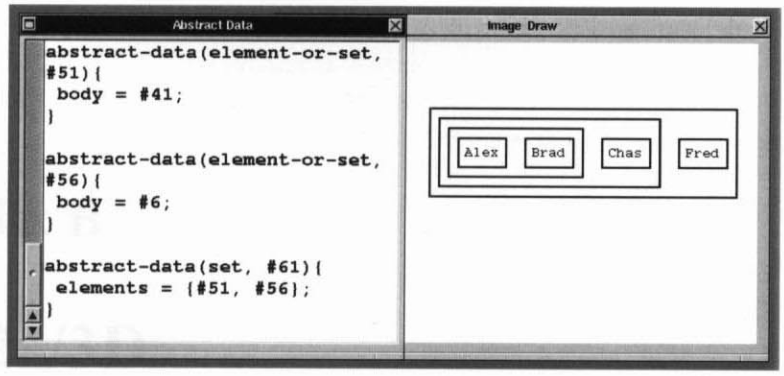


Figure A.13: IMAGE — set diagram screenshot.

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.

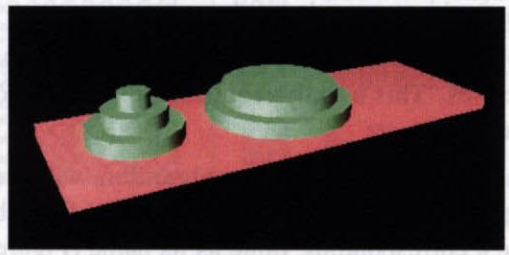


Figure B.2: 3D tower of Hanoi.

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.

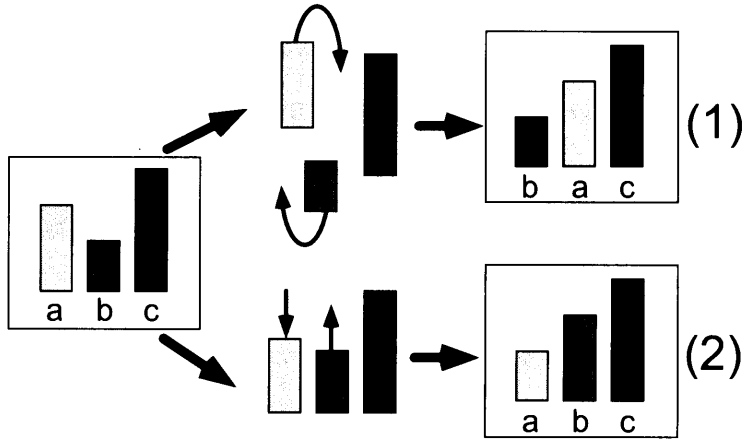


Figure B.3: Two ways of naming objects.

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.

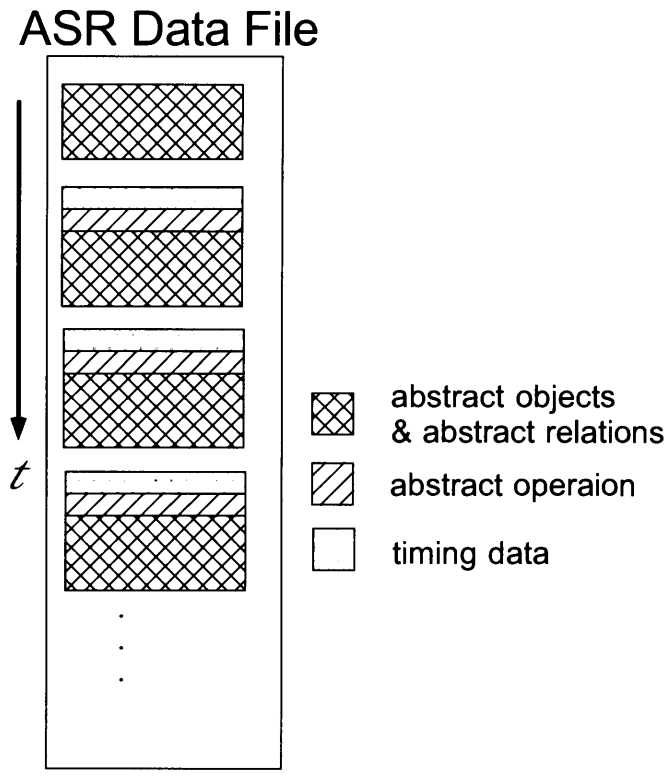


Figure B.4: ASR data file.

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

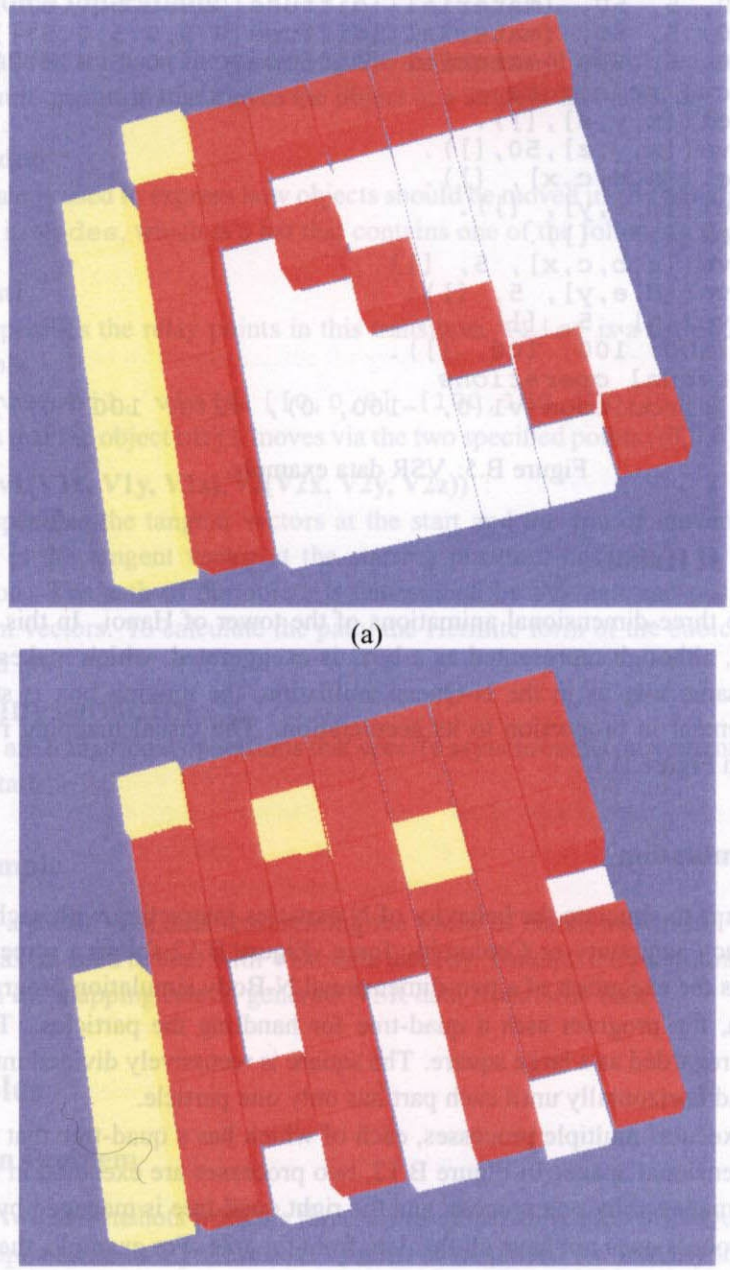
The N-Body problem is to simulate the behavior of N particles interacting with each other through a long-range force such as gravity or Coulombic force. Figure B.12 shows a screenshot from the animation that depicts the execution of a two-dimensional N-Body simulation program.

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.

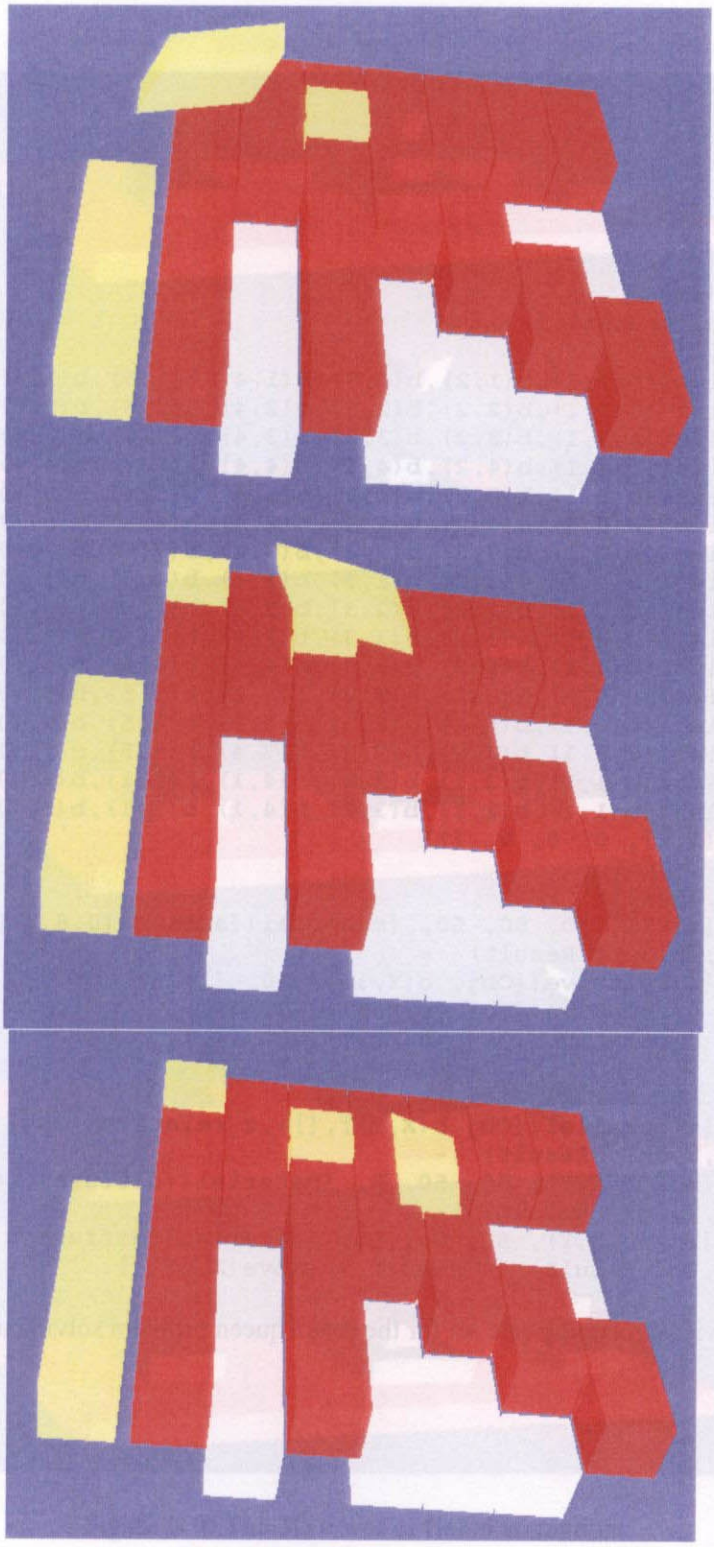


Figure B.7: 7 queen problem — distortion technique.

```
:- 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.

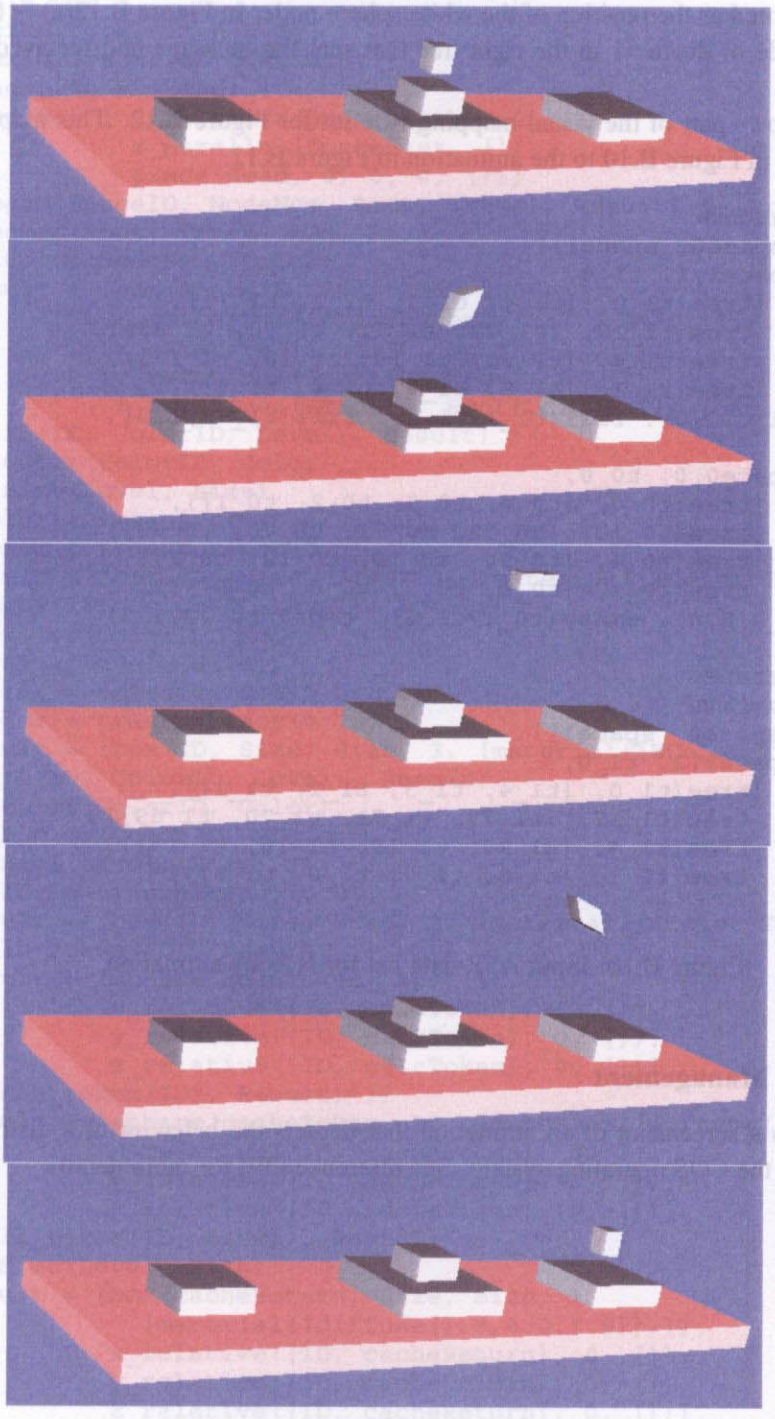


Figure B.9: The 3D tower of Hanoi animation.

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])]).
...
```

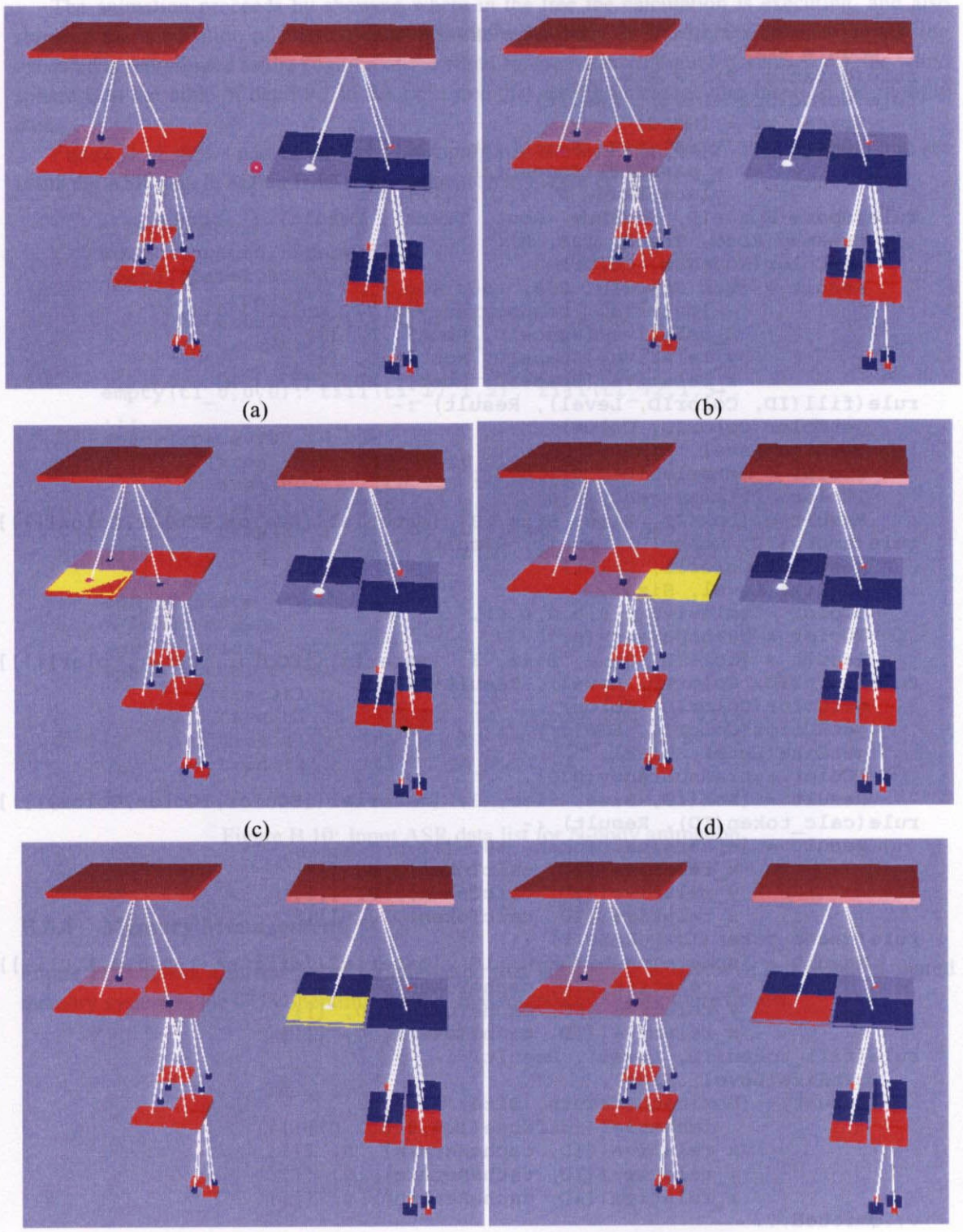
Figure B.10: Input ASR data list for N-body animation.

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.

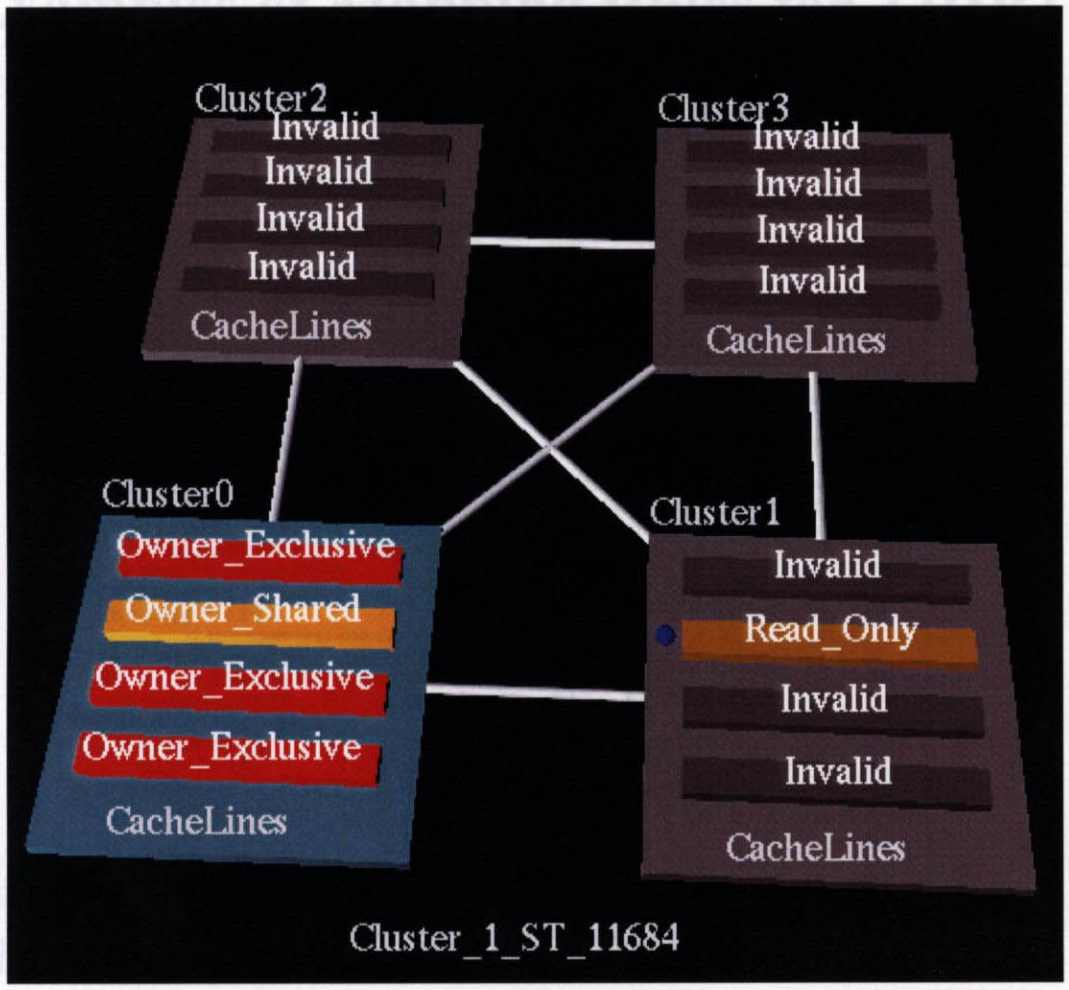


Figure B.13: An example of 3D visualization: cache.

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