Chapter 6

Visualizing and Browsing Constraints in Visualization Rules

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

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

6.1 Introduction

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

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

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

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

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

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

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

- Degrees of freedom of objects are represented by animation. Constraining objects decreases their degree of freedom.
- Whether an attribute has any degree of freedom can be represented by its display. An attribute with a degree of freedom is animated so that its various valid values are shown to the user. A constrained attribute is animated as appearing to satisfy the constraints.
- An attribute that has no degrees of freedom, i.e., an attribute that has a determined value, is animated (jerked or twitched) as if it is trying to change its value but cannot. Cartoon animation techniques are used to represent this situation.
- Attributes that are related are animated together. That is, attributes that have no relation to each other should not be animated together.

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

6.2 Browsing Three-Dimensional Constraint Graphs

6.2.1 Basic Representation

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

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

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

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

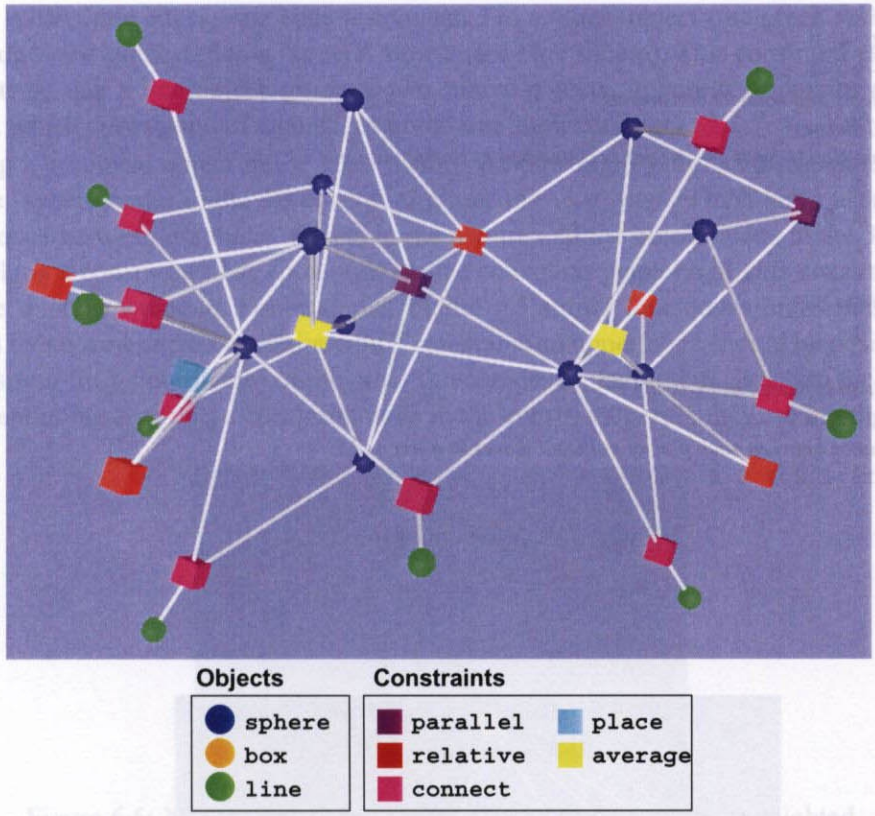

Figure 6.1: The constraint graph corresponding to the VSR data in Figure 6.2.

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

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

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

Figure 6.2: VSR data for a tree.

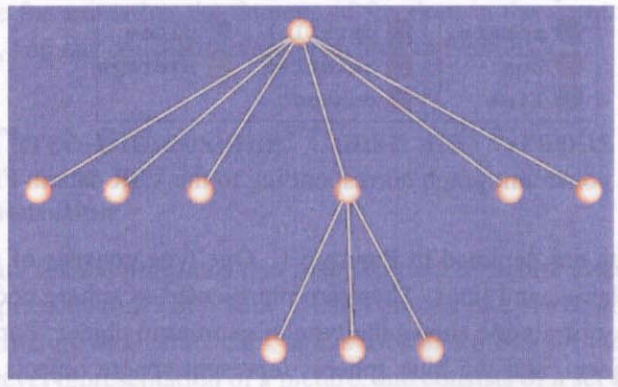

Figure 6.3: The resulting picture corresponding to Figure 6.2.

constraints.

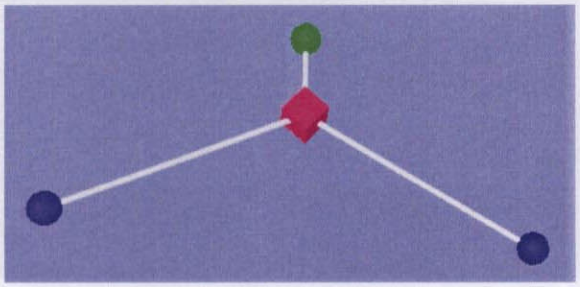

Figure 6.4: A connect constraint (purple box).

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

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

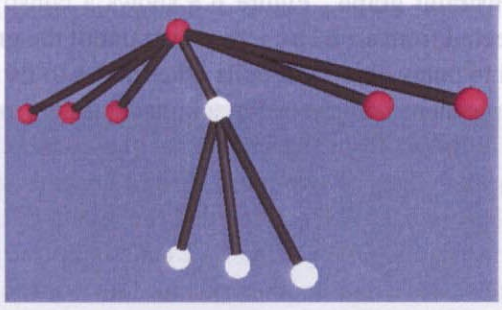

Figure 6.5: Nodes constrained by the clicked constraint are highlighted.

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

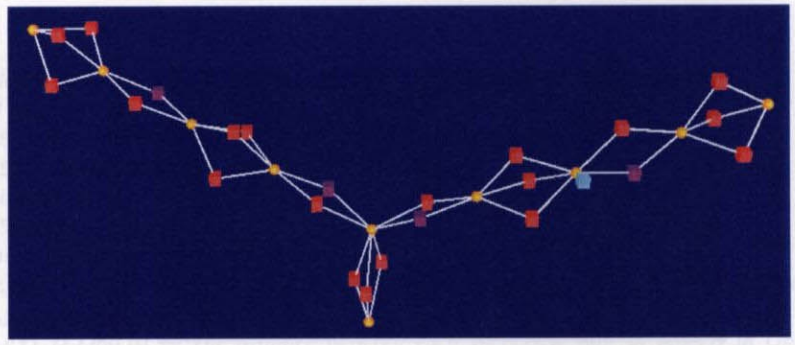

Figure 6.6: A constraint graph with regular structure.

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

6.2.2 Changing Layout to Explore Constraint Graphs

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

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

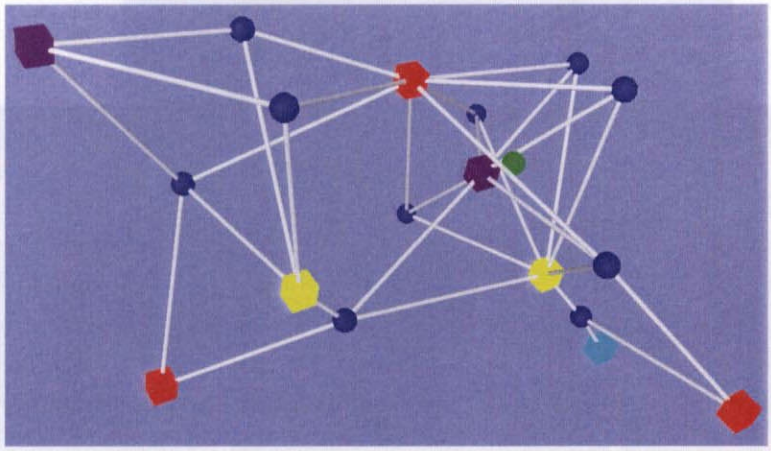

Figure 6.7: A constraint graph of a tree in Figure 6.5 (without line constraints).

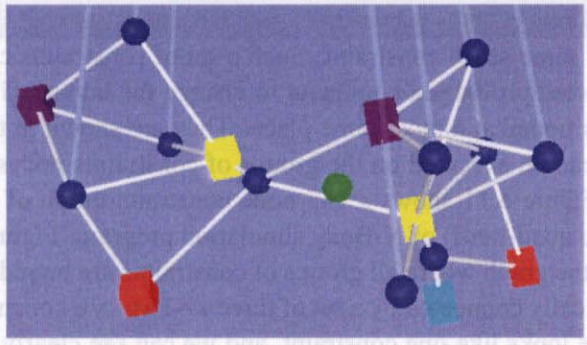

Figure 6.8: A constraint graph — a `relative` constraint is set aside.

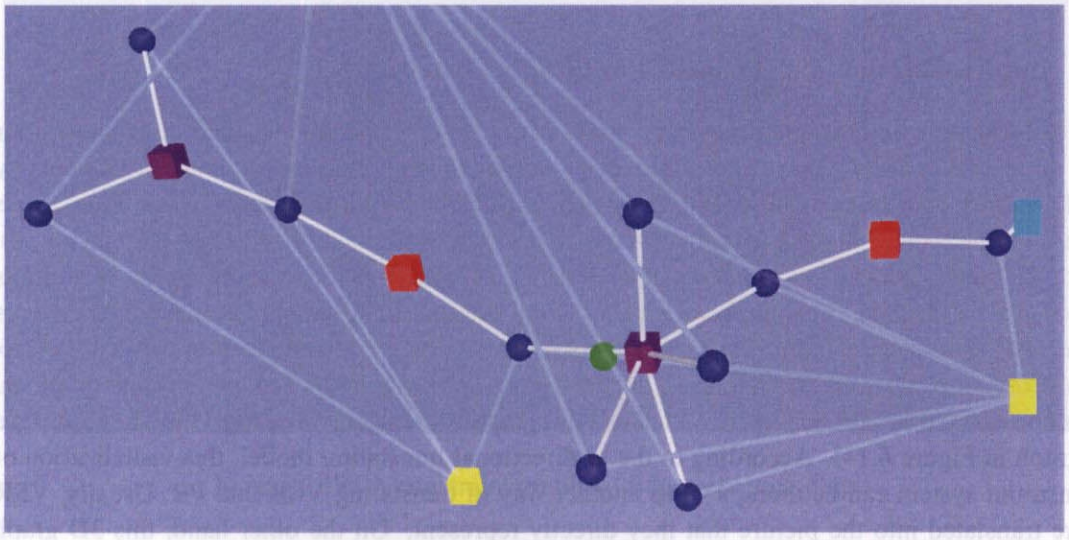

Figure 6.9: A constraint graph — a `relative` and two `average.` constraints are set aside

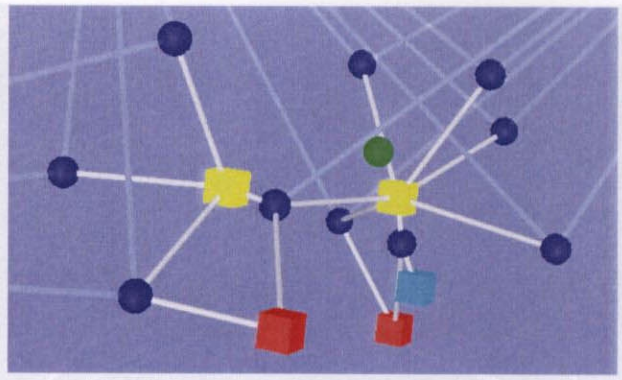

Figure 6.10: A constraint graph — a `relative` and two `parallel` constraints are set aside.

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

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

6.2.3 Implementation of 3D Constraint Graph Visualizer

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

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

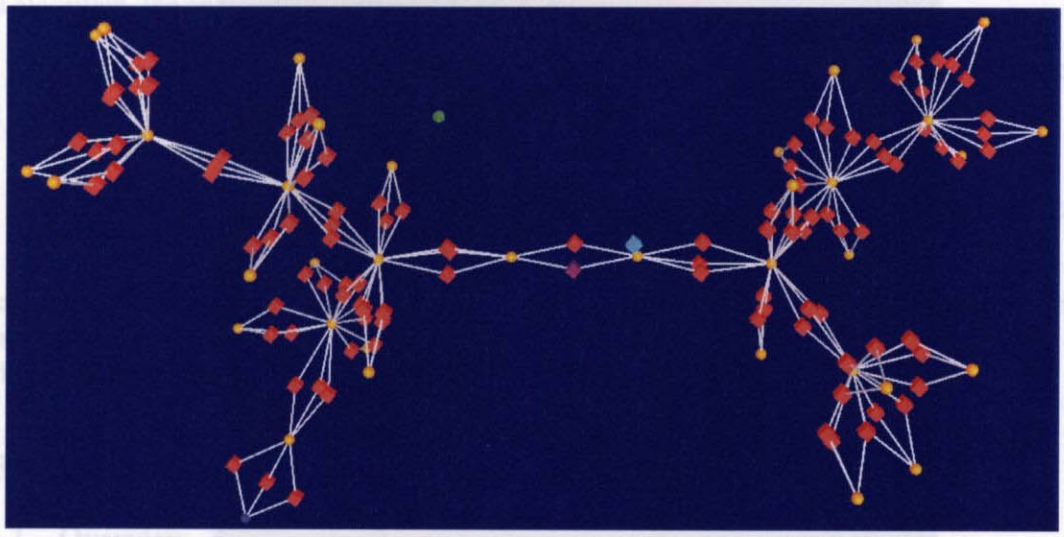

Figure 6.11: A constraint graph of N-body animation.

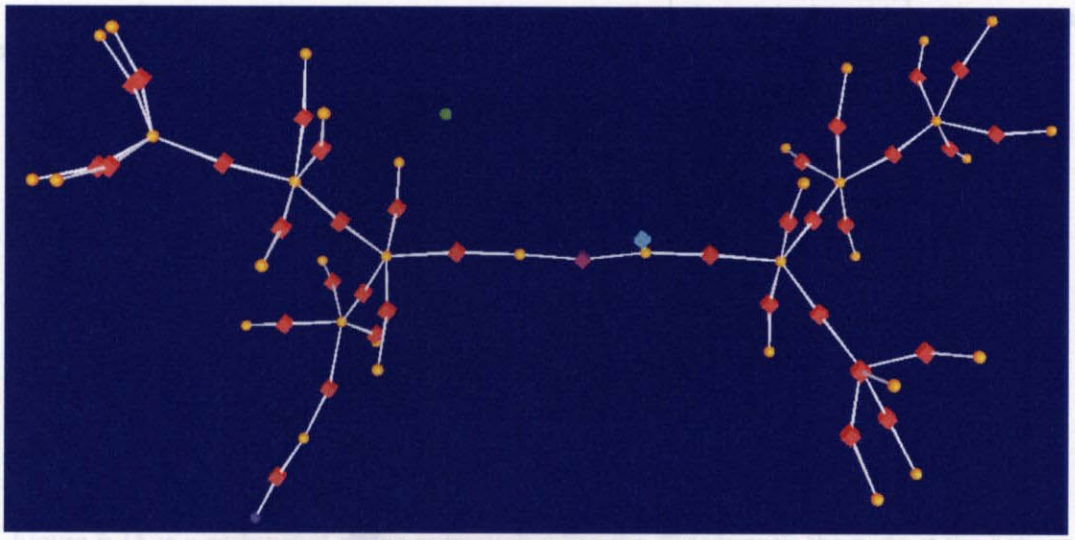

Figure 6.12: A constraint graph — Edges are bound up.

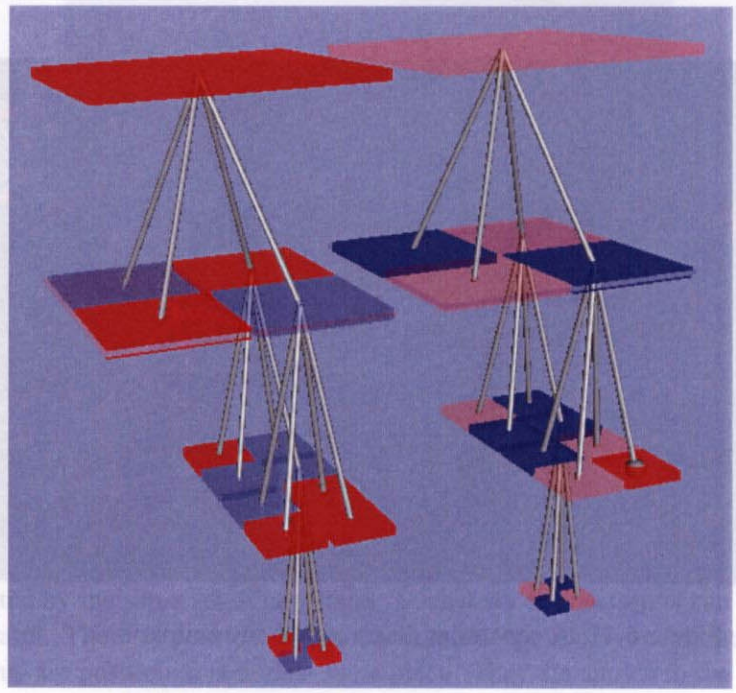

Figure 6.13: Target picture — Two quad-trees.

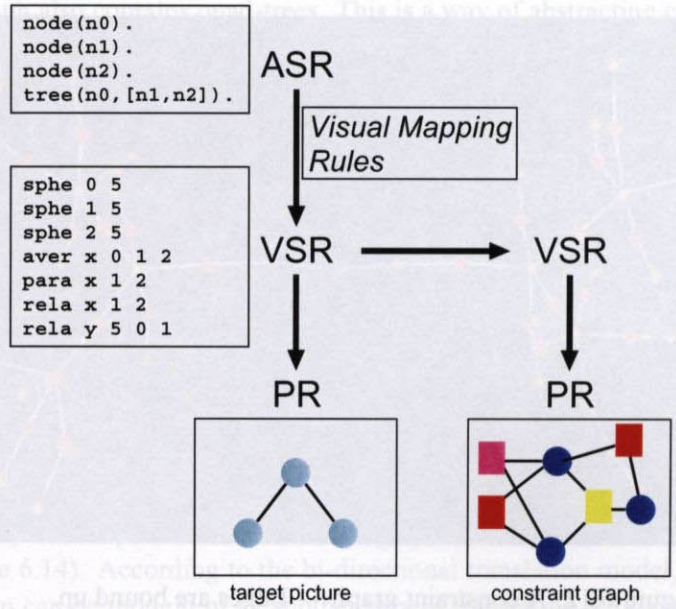

Figure 6.14: An example of translating ASR into PR via VSR.

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

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

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

6.3 Animating Freedoms in a Constraint System

6.3.1 Overview

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

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

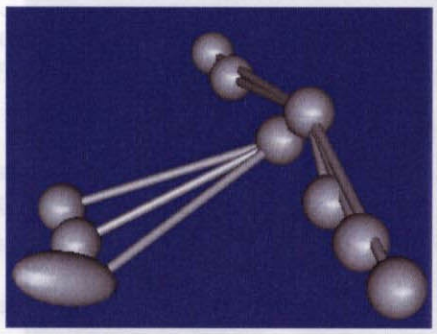

Figure 6.15: Pulling a node in a tree — Movable in this direction.

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

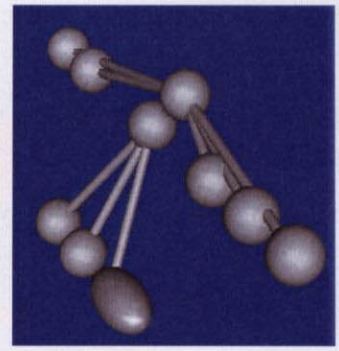

Figure 6.16: Pulling a node in a tree — Immovable in this direction.

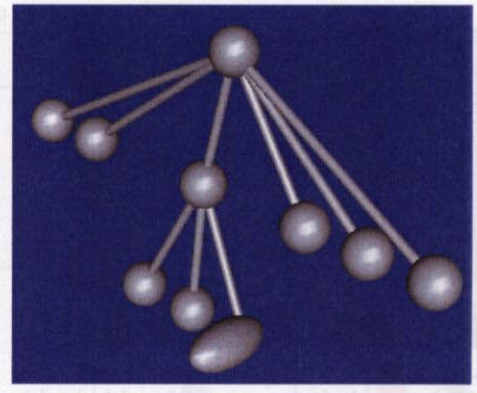

Figure 6.17: Pulling a node in a tree — All nodes are well-constrained.

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

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

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

6.3.2 Implementation of the Freedom Viewer

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

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

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

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

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

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

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

4. The system solves the modified constraint system again.

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

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

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

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

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

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

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

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

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

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

6.4 Related Work

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

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

6.5 Concluding Remarks

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

Chapter 7

Conclusions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

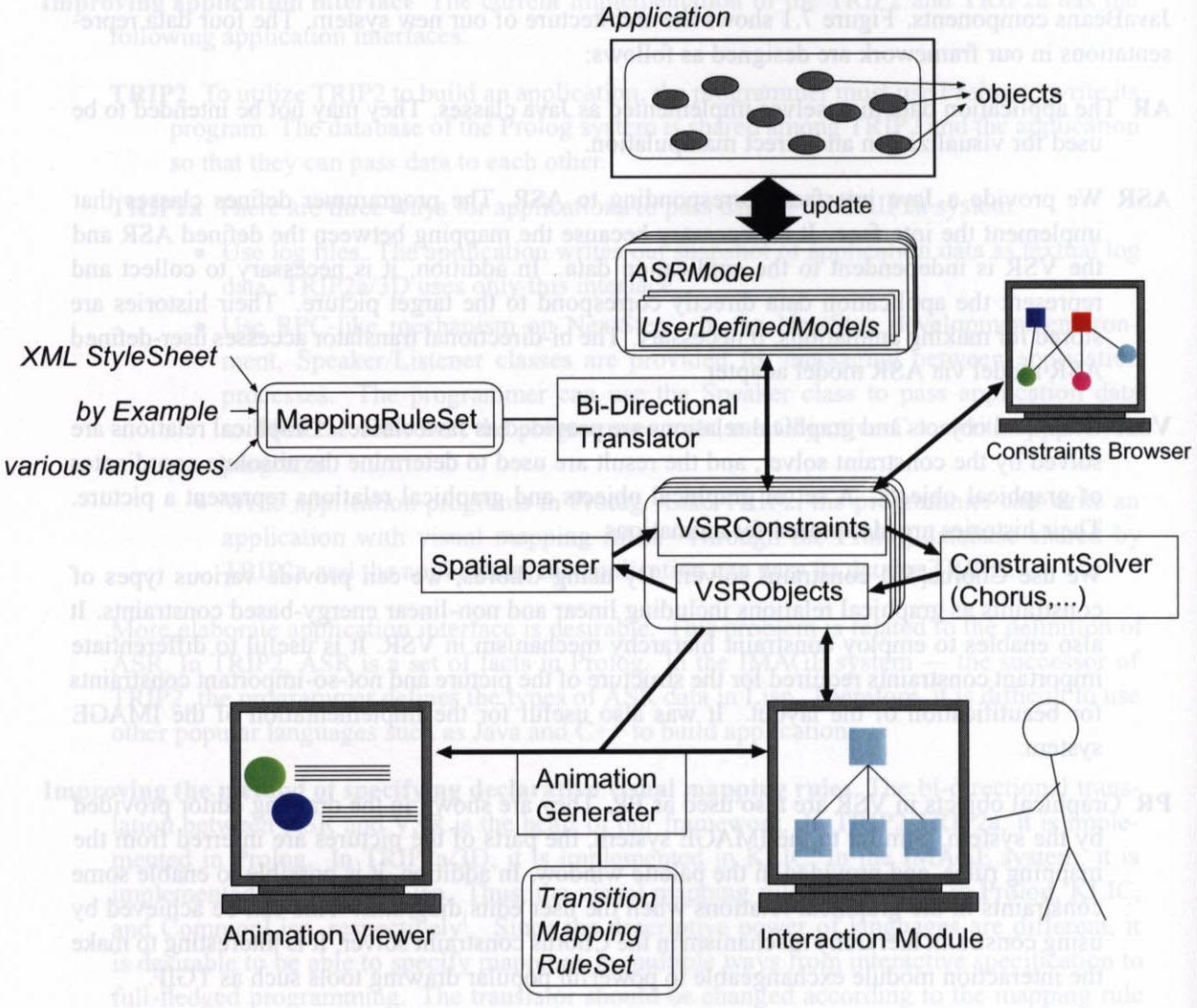

Figure 7.1: The architecture of the next system.

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

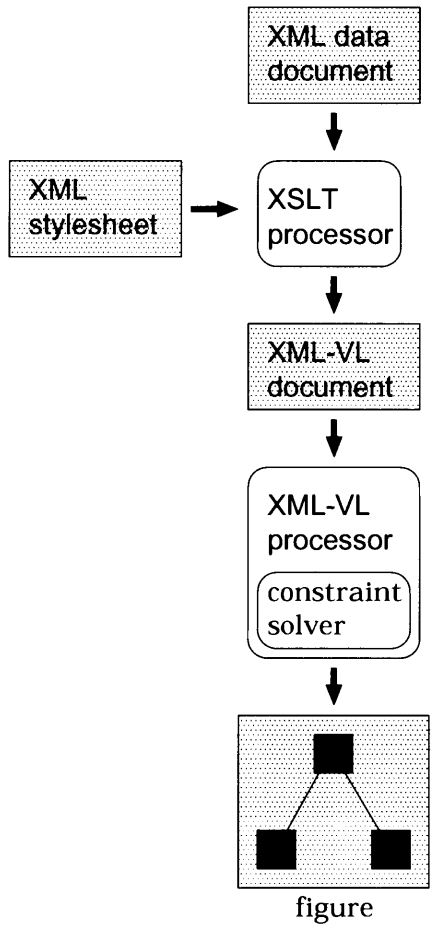

Figure 7.2: Visualization framework with XML-VL[54].

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

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

visual and interactive communication between users and computers.