A General Framework for Bi-Directional Translation between Abstract and Pictorial Data

Shin Takahashi Satoshi Matsuoka Akinori Yonezawa*
Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 Japan

Tomihisa Kamada†
Research and Development, ACCESS CO., LTD.
1-7-1 Sarugaku-cho, Chiyoda-ku, Tokyo, 101 Japan

Abstract

The merits of direct manipulation(DM) are now widely recognized. However, DM interfaces incur high cost in their creation. To cope with this problem, we present a model of bi-directional translation between internal abstract data of applications and pictures, and create a prototype system TRIP2 based on this model. Using this model, general mapping from abstract data to pictures, and from pictures to abstract data, is realized merely by giving declarative mapping rules, allowing fast and effortless creation of DM interfaces. We also apply the prototype system to the generation of the interfaces for kinship diagrams, graph diagrams, and an Othello game.

keywords : Bi-Directional Translation, Direct manipulation, User Interface, User Interface Management System, Visualization

1 Introduction

Graphical and interactive user interfaces based on *direct manipulation*(DM)[27], allowing the users to point to, grasp, and drag the real objects on the screen, are now standard. To ease the burden of high cost of its creation, we employ User Interface Management Systems (UIMSs) such as NextStep[24], and user interface toolkits such as MacApp[1] and InterViews[29]. However DM style interfaces are still costly, because UIMSs and toolkits usually only provide 'canned' abstractions of behaviors for a fixed set of simple graphical interface objects such as buttons and scrollbars. Such a fixed set of abstractions may be sufficient for simple, dialogue-style interfaces, but for cases where the semantics of complex application data must be reflected and be manipulable on the display, the programmers usually do not necessarily have the appropriate graphical and input abstractions available. This is primarily because current models of user interfaces are usually the models of the interaction architecture, and lack the support for consistent framework that allow visualization and manipulation of high-level abstract data, i.e. semantics of applications.

In our previous research, TRIP[10][12] realized general visualization of abstract application data. It is based on the model of *translation* from application's data representations into pictorial representations. In this model, abstract objects and relations in abstract concepts represented as application's data are mapped to graphical objects (such as boxes and circles) and graphical relations (such as horizontal/vertical listings and line connections). The mapping is specified not with procedures, but with declarative mapping rules.

In this paper, we extend this one-way visualization framework to bi-directional translation between application's data representations and pictorial representations of the user interface. With this framework, the feedback of user's manipulation can be generally achieved via inverse translation from picture data into abstract data. The specification of inverse translation is achieved (almost) with no extra cost, because (almost) identical rules can be used for mappings in both directions. We have implemented TRIP2, a prototype UIMS based on this model. In TRIP2, direct manipulation interfaces for various types of data can be achieved just by specifying declarative mapping rules.

2 The Bi-Directional Translation Model

2.1 Four Types of Data Representations

The idea of bi-directional translation via direct specification of mappings between application and picture data is conceptually clean and simple. The difficulty is, however, in its applicability as a general-purpose schema. Since there are infinite kinds of both application's data representations and pictorial representations, ad-hoc schemes force the users to specify representation-dependent translation rules for every new combinations of abstract and picture data.

To resolve this problem, we employ two intermediate, universal representations, so that mapping rules have to be specified only for translations between those representations. One is *Abstract Structure Representation (ASR)*, and the other is *Visual Structure Representation (VSR)*. The intermediate representations serve as *pivots*, so that when either the abstract or pictorial representations change, the mapping rules between ASR and VSR are essentially unaffected.

Altogether, the translation process handles the following

*Email: {takahasi,matsu,yonezawa}@is.s.u-tokyo.ac.jp
†Email: tomy@access.co.jp

In the Proceedings of the ACM User Interface Software and Technology '91 (UIST'91), November 11-13, South Carolina.

four representations:

Application's Data Representation(AR) This is the application-specific data representation. It may thus take any form of representation. They are, for example, natural languages, program lists, and form data for a database.

Abstract Structure Representation(ASR) This represents the underlying abstract structure, which is a set of *relations* among *abstract objects* that are extracted from application's data. Application's data are analyzed and translated into this universal representation. In the inverse translation process, the original application's data representation is generated from this representation.

Visual Structure Representation(VSR) This represents the underlying structure of a picture, which is a set of *graphical relations* among *graphical objects*. In VSR, no absolute coordinates are specified. Similar to the abstract structure representation, this representation is translated into and from the pictorial representation.

Pictorial Representation(PR) This is the representation of the picture to be rendered directly on display devices. The users look at and manipulate pictures generated from this data.

2.2 Visual Mapping and Inverse Visual Mapping

In our model, the translation process from/into application's data into/from picture data proceeds as outlined below. For formal details, see [10].

Firstly, the original application's data representation is analyzed, and its ASR data are generated. Analyzers are customized for each kind of application data, since they are dependent on the semantics of application data.

Secondly, the ASR is translated into the VSR, which is a process we refer to as *visual mapping*. Abstract objects are mapped to graphical objects, and abstract relations are mapped to graphical relations among the corresponding graphical objects. The users can create a variety of pictures merely by changing the mapping rules. The inverse translation from the VSR into the ASR is referred to as *inverse visual mapping*. The details of the mapping rules are described in the ensuing sections.

We note that the mapping rules are declarative, and thus do not embody directional information of translation within the rules. As a result, a single rule could be employed both for visual mapping and for inverse visual mapping. By contrast, if we were to specify the mappings procedurally, it would be necessary for us to program two totally distinct functions for each direction.

Finally, the VSR is translated into the target pictorial representation (PR). The graphical relation data are first translated into geometrical constraints among the *picture objects*. To determine their actual positions (x- and y-coordinates) in a display space, the constraints are solved by constraint solvers. Then, the device-dependent picture data are generated, which are rendered on the display. In the inverse translation, the VSR is generated from the picture data by performing *spatial parsing* on the picture; the details are described in Section 3.4.

To summarize the inverse translation, first, picture data are analyzed with the spatial parser, and the VSR is generated; Second, the VSR is mapped to the ASR via "inverse visual mapping"; Finally, application's data are generated from the abstract structure representation with an application specific synthesizer.

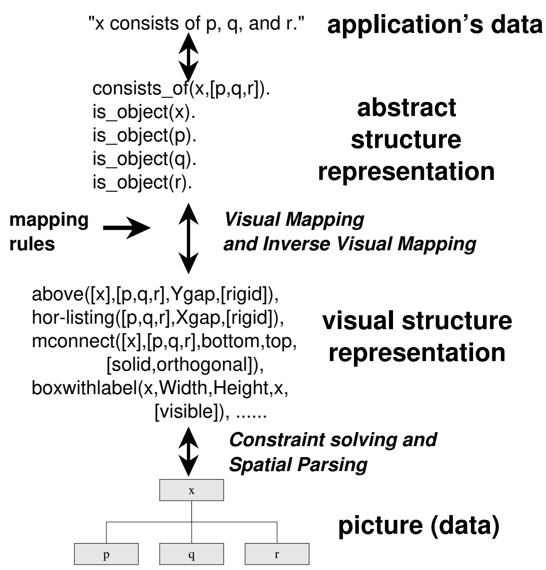


Figure 1: An Example of the Model

Figure 1 illustrates an example of the four types of representations. In this example, the application's data is represented as a natural language sentence "x consists of p, q, and r". The corresponding ASR data are consists_of(x, [p, q, r]), and is_object(x), is_object(p), etc. The consists_of is an abstract relation, and is_objects are abstract objects. The corresponding VSR data are above([x], [p, q, r], Ygap, [rigid]), hor-listing([p, q, r], Xgap, [rigid]), mconnect([x],[p,q,r],...), boxwithlabel(x, Width, Height, x, [visible]), etc[1]. The above, hor-listing, and mconnect are graphical relations, and the boxwithlabels are graphical objects. As a pictorial representation, a tree picture is generated in Figure 1.

3 The TRIP2 System

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

3.1 System Overview

Figure 2 shows an overview of the TRIP2 system. TRIP2 evolved from TRIP (TRanslation Into Pictures)[10, 12], which handled only one-way translation from abstract data into picture data. The new features of TRIP2 that realize our bi-directional translation model are as follows:

1. The **interaction module** which allows the users to manipulate the pictures directly with input devices such as a mouse. Currently, this module provides the users with a freehand drawing interface similar to

[1] The VSR here denotes the following structure: The object x is placed above the objects p, q, and r. The objects p, q, and r are horizontally arranged; The object x is connected to the objects p, q, r by lines. The object x is a box with a label x.

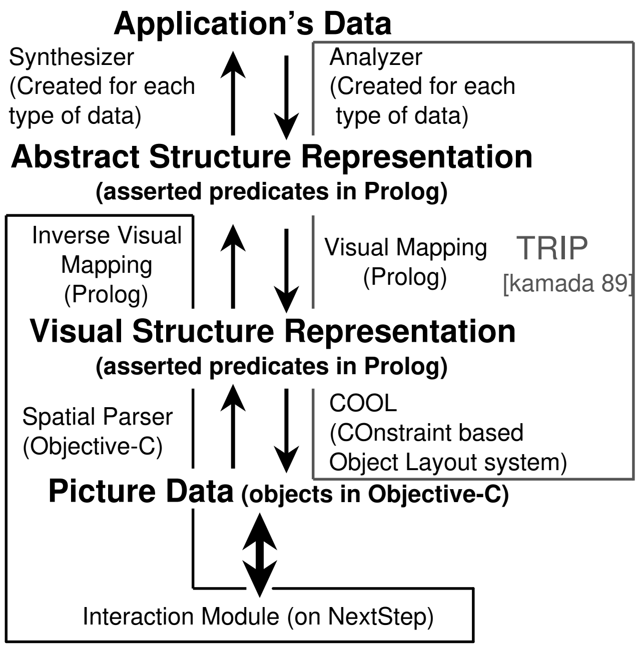


Figure 2: The TRIP2 System

MacDraw[2]. This module is implemented on NextStep by using Objective-C; thus, low-level graphical interactions, such as selection of a picture object with a mouse, are treated as method invocation of Objective-C objects that represent the picture.

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

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

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

The users interact with TRIP2 in the following way:

1. Compile the mapping rules, and read them into the rule database.
2. Read the ASR data from a file into the *data* window.
3. Execute TRIP to translate the data into a picture.
4. Edit the picture in a freehand manner.
5. Execute the *flush* command. This command first translates VSR data into ASR data, and then re-executes TRIP , i.e. translates the new ASR data into a picture. As a consequence, application semantics-directed beautification is achieved.
6. Return to 4.

[2]MacDraw is a trademark of Claris, Inc.
[3]SB-Prolog version 3.1, Copyright (C) 1986 at Stony Brook; 1987 University of Arizona.

Figure 3 shows an example of using TRIP2. Each screen dump of left-right windows shows the data of a Japanese family as ASR and its corresponding picture as a kinship diagram (Figure 3(a)). The left window is the *Data Window* which displays ASR data, and the right window is the *ObjView Window* which displays its pictorial representation. The users are free to modify either data at any time, although they interact only with the pictures in the *ObjView Window* in this example. Now, suppose that a new member, *Midori*, joins the family by marrying *Shinichi*. To express this fact, we draw a rectangle (Figure 3(b)), write the label *Midori*(Figure 3(c)), and connect it to the rectangle labeled with *Shinichi* by a line(Figure 3(d)). (Note that this label *Midori* was not produced by an explicit command to label the rectangle; rather, it is a textual picture object that has an equal status to the rectangles and lines just drawn.) Then, we execute the *flush* command. These modifications are translated back into ASR data, and the kinship diagram is redrawn from the new data (Figure 3(e)). Now a baby, *Tomomi*, is born: We draw a circle between two rectangles, draw a new rectangle, write the name, and connect the circle and the rectangle (Figure 3(f)). Finally, we execute the *flush* command. These modifications are fed back, and the diagram is regenerated (Figure 3(g)).

3.2 The TRIP Module

The TRIP[10, 12] module is responsible for the translation from abstract data into concrete pictures. As illustrated in Figure 2, the TRIP module consists of three submodules (1) application's data analyzer, (2) the visual mapper, and (3) a layout system called COOL (COnstraint-based Object Layout system).

The analyzer translates application's data into ASR data, which is a set of Prolog predicates. In [12], examples of several types of application's data are presented, such as English sentences representing kinship relations, S-expressions in Lisp, and C programs. Those analyzers are not integral part of TRIP2, but TRIP2 provides an editing window for ASR data as shown in Figure 3. Thus, the users can interactively manipulate both ends of application-independent representations at the same time.

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

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

and the visual mapping rules:

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

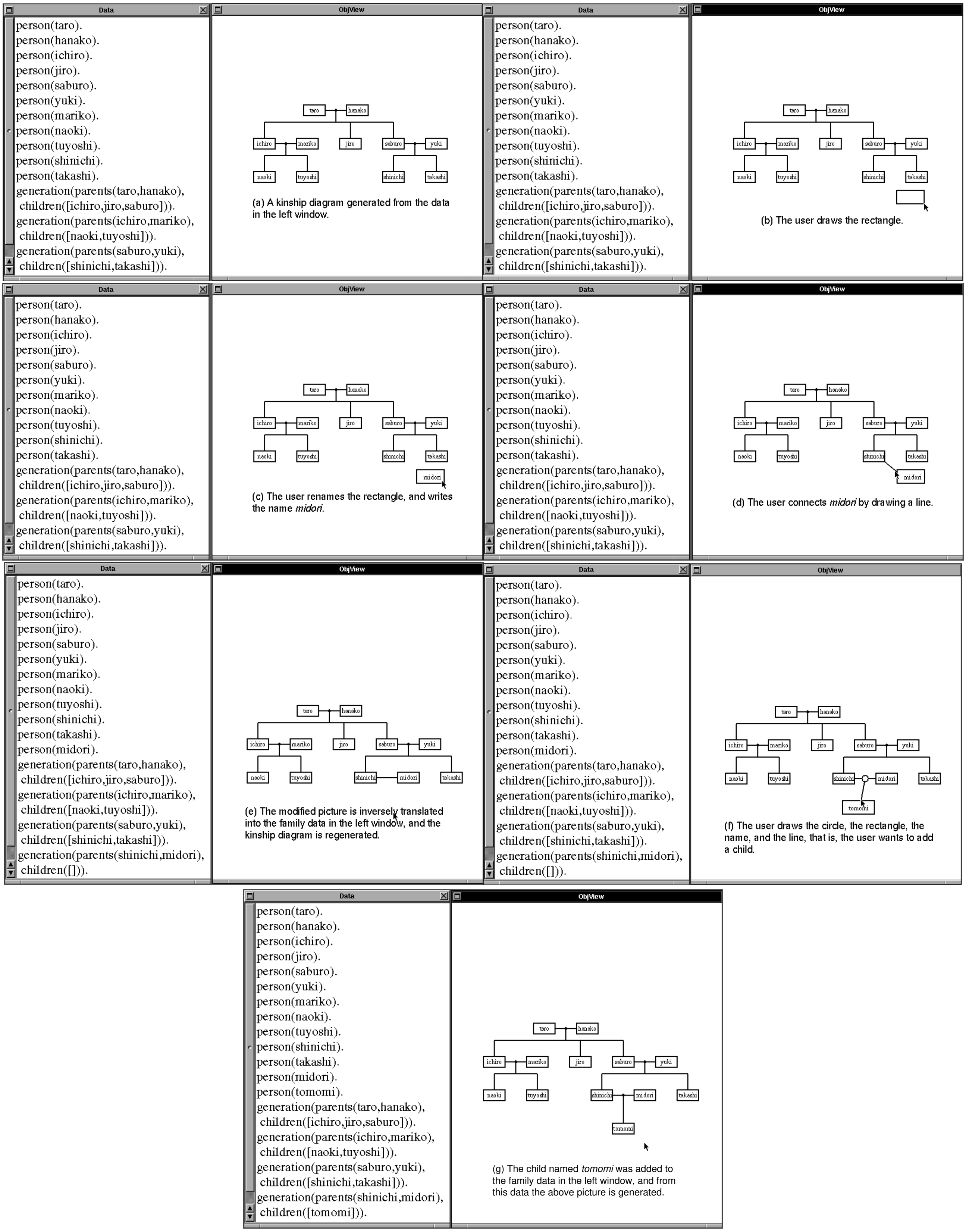


Figure 3: A Kinship Diagram

```
        % label 'NAME'.
        label(NAME, NAME, []).

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

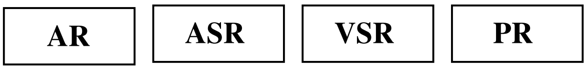


Figure 4: The Generated Picture

the picture in Figure 4 is generated. To execute the visual mapping, TRIP calls the predicates listed in the argument of predicates `objectmap` and `relationmap`. The predicates of the `objectmap` maps abstract objects to graphical objects, and the predicates of the `relationmap` maps abstract relations to graphical relations. In this example, the predicate `itemmaping` repeatedly applies the predicate `itemmap` to the abstract data "`item`"s, and the predicate `itemmap` maps "`item(NAME)`" to a box with a label `NAME`. The predicate `ordermap` lays out the "`item`"s horizontally.

Finally, VSR data are translated into picture data by COOL. In COOL, VSR data are declared as special predicates that generate (1) drawing data, and (2) constraints on the geometric attributes in the drawing data. For example, the predicate `box` generates data for drawing a rectangle, and constraints on coordinate variables of the rectangle. The predicate `horizontal` generates constraints on y-coordinate variables of picture objects corresponding to the graphical objects listed in its argument. These constraints are solved by the constraint solvers in COOL. The constraint solver can handle two types of constraints: "rigid" constraints, which must be solved exactly, and "pliable" constraints, which may be solved approximately by the least square method. The purpose of pliable constraints is to allow COOL to handle over-constrained systems. In addition, COOL has special predicates and another constraint solver for visualizing undirected graph structures as balanced graphs. The solutions computed by the constraint solvers instantiate the coordinate variables in the drawing data. They are then passed to the interaction module to generate the Objective-C picture objects.

3.3 The Interaction Module

The users of TRIP2 manipulate pictures directly via the interaction module which provides the generic lowest-level editing facility. They can create, move, and delete picture objects such as rectangles, circles, etc. We stress that the resulting interface is NOT a structured editor for some specific data — rather, the users simply draw rectangles or circles as if they were using freehand drawing systems such as MacDraw. This is UNLIKE graphical editors which are customized for each application, such as Unidraw[31].

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

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

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

3.4 Implementation of the Inverse Translation

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

Translation from PR into VSR — Spatial Parsing Here, we refer to translation from PR into VSR as *spatial parsing*. Although spatial parsing has been studied (for example, [7, 14, 15]), its application to GUI poses difficulties due to the requirement of real-time use, because there are a great number of graphical relations with no a priori information in a picture.

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

To realize real-time responsiveness, we use an *incremental* spatial parser, which maintains the graphical relations and updates only its modified portions. Our incremental spatial parser is invoked when the users create or delete a picture object.

- **When a picture object is deleted:** The graphical object (in VSR) corresponding to the deleted picture object (in PR) is removed from each (existing) graphical relation in which it is contained. For example, when the rectangle with a label `p` in Figure 1 is deleted, `p` is removed from the `above`, `hor-listing`, and `mconnect` relations. To achieve such modifications efficiently, the interaction module maintains a table that maps each graphical object to a list of graphical relations in which it is contained.

- **When a new picture object (obj) is created:** The message `addWithTest:obj` is sent to each existing graphical relation object. The invoked method tests whether the `obj` is 'addable' to the graphical relation, and if so, adds it to the graphical relation. For example, if a new rectangle is drawn with a label `s` on a level of `p`, `q`, and `r` in Figure 1, `addWithTest:s` message is sent to each graphical relation object, `above`, `hor-listing`, and `mconnect`. Because `x` is above `s`, `s` is addable to the relation `above`. Because `s` is horizontal with `p`, `q`, and `r`, it is also addable to the relation `hor-listing`. However, since a line connecting `x` and `s` is nonexistent, `s` is not addable to the relation `mconnect`.
- **When a picture object is moved:** The spatial parser is also invoked when a picture object is moved; this change is treated as a combination of deletion and addition.
- **Adding a new graphical relation:** Although the above scheme can handle the modification of existing graphical relations with an arbitrary number of graphical objects, the shortcoming is that it cannot handle the creation of new graphical relations. For example, the `connect` relation, which represents two graphical objects connected by a line, must be created when the users draw a line that connects two picture objects that were previously unrelated. We alleviate this by associating generations of new graphical relations with drawings of the picture objects. For example, when a line connecting two objects is drawn, a `connect` relation is created, and when an arrow is drawn, an `arrow` relation is created. We are working now to provide other means of generating other types of graphical relations.

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

Ideally, we should be able to use identical rules for visual mapping as well as for inverse visual mapping, because declarative rules do not have 'directional' information. Due to technical restrictions of the implementation of our current prototype, however, the users must provide separate rules for visual mapping and inverse visual mapping. Fortunately, the structure of the rules are essentially identical; thus, an experienced user will be able to derive one from the other with ease. For example, inverse visual mapping rules for the Figure 4 are as follows:

```
invomap([inv_itemmapping]).
invrmap([inv_ordermapping]).
inv_itemmapping :-
    % finds 'item' data
     inv_itemmap(NAME),
    % writes 'item' data to an output file
     write('item('),
     write(NAME), write(').'), nl,
    % tries for another data
     fail.
% 'item' data is a
inv_itemmap(NAME) :-
    % box with its name 'NAME',
     box(NAME, _, _, _),
    % and the label 'NAME'
     label(_, NAME, _).
inv_ordermapping :-
    % finds 'order' data
     inv_ordermap(ILIST),
    % writes 'order' data to an output file
     write('order('),
     write(ILIST), write(').'), nl.
% 'order' data is a
inv_ordermap(ILIST) :-
    % horizontal ordering
     horizontallisting(ILIST, _, _).
```

These rules (1) generate `item(NAME)`s which are derived from box and label objects in the VSR database, and (2) generate `order(ILIST)` which is derived from the horizontallisting relation in the VSR database, then (3) output them in a certain format. Compared to the visual mapping rules for Figure 4, we can see that each clause corresponds one-to-one, and the predicates in the clause body and their ordering are almost the same. Derivation of other inverse mapping rules are similar.

In this example, the predicate `inv_itemmap` has essentially the same structure as `itemmap`, but width and height of the `box` are ignored. This would allow boxes of an arbitrary size. In the cases when one desires to restrict the size of boxes within a certain range, predicates should be added regarding the width and height:

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

4 Examples

In this section, we show the examples of TRIP2 in action. All the figures in this section show the screen dumps of two windows of TRIP2. In the following examples, we focus on the feedback of user's operations on the PR data.

4.1 A Simple Graph Editor

This example is a simple graph editor on TRIP2. The visual mapping rule for this example uses the `adjacent` relation which is an entry to the graph layout constraint solver in TRIP2. In this mapping rule, nodes are mapped to circles, and arranged by the graph layout constraint solver. Edges are represented as straight lines. The graph drawing algorithm used here is described in [10, 11].

4.2 A Small Othello Game

This example provides an interface for a small Othello game application. Figure 6 shows the screen dumps of the windows during the game, which display the board data (ASR) and its pictorial representation. The Othello application itself is written in Prolog, and uses this ASR data directly as its representation of the board (therefore, there is no translation process between AR and ASR). The board data consists of six rows of six characters (Figure 6(a)). The character *e* means that the position is empty. The characters *b* and *w* represent black and white stones respectively.

Player-1 first draws a circle with its color attribute black, which represents a black stone (Figure 6(b)), and executes the *flush* command. The modified picture is translated into the board data. Then, the application reverses the intermediate stones. The resultant ASR of the board is again visualized to update the PR (Figure 6(c)). Player-2 then draw a white stone (Figure 6(d)). This modification is also fed back to the board data (Figure 6(e)).

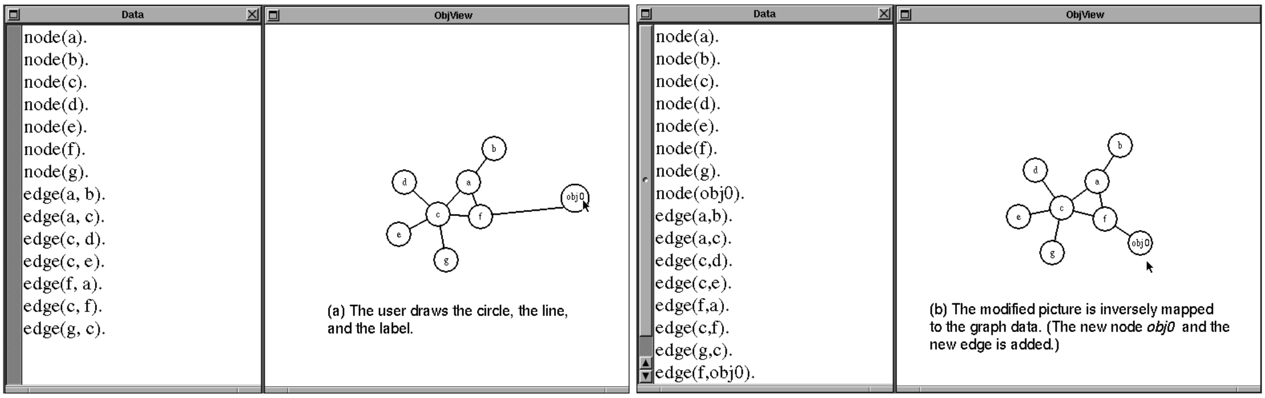


Figure 5: A Simple Graph Editor

5 Related Work

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

Visualization of *numerical* data has been studied intensively: Scientific visualization is now one of the most important areas of numerical data visualization — powerful personal tools such as *Mathematica*[32] are now in widespread use. Business graph packages are also used extensively in office and personal work. One interesting work is by Mackinlay[16], in which effective graphical presentations (such as bar charts, scatter plots) of relational information are automatically combined and designed. Although TRIP2 is not primarily intended for numerical visualization[4], a similar issue would arise as to which visual mapping rules should be selected and combined to provide an effective interface for complex application data. We are examining this topic as one of the current research issues.

There are a number of researches on visualization of non-numerical data as well. For example, visualization of programs [13, 22, 30], and visualization of database schemes [33] has been studied. Here, the advantage of TRIP2 (and TRIP) is that it is a general-purpose system that can be applied to numerous types of visualization simply by defining declarative mapping rules. Also, in the context of TRIP, general visualization of abstract data and relations that can be mapped to ordering relations among the constituent objects is presented in [21].

There are several researches on establishing general schema for feedback of user's operations on pictures. In the area of graphical UI, visual and/or demonstrational specification of how the primitive interface objects should look, feel, and work is possible with systems such as Peridot[19], and Interface Builder[24]. *Visual programming* systems also recognize user's operations, but are special-purpose in the sense that the interfaces are usually tailored for a single (visual) language; they are surveyed in [3, 28]. Of special interest are GRAFLOG[25] which is an interactive graphics interface in which drawings receive linguistic interpretations, i.e., a certain kind of visual language, PAM[15] which is a system for manipulating text-graphic patterns, and PLG[7] which is a grammar for specifying the syntax of visual languages. Constraint-based graphical UI systems such as Juno[23], ThingLab[2], ThingLabII[17], Metamouse[5] allow the users to specify the constraints among the picture objects, but do not explicitly support the specifications of mappings between application and picture data. Advanced UIMSs such as Garnet[20] and Unidraw[31] provide the support for visual interfaces which reflect application semantics, but the schema includes more procedure-oriented and/or library-based specifications.

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

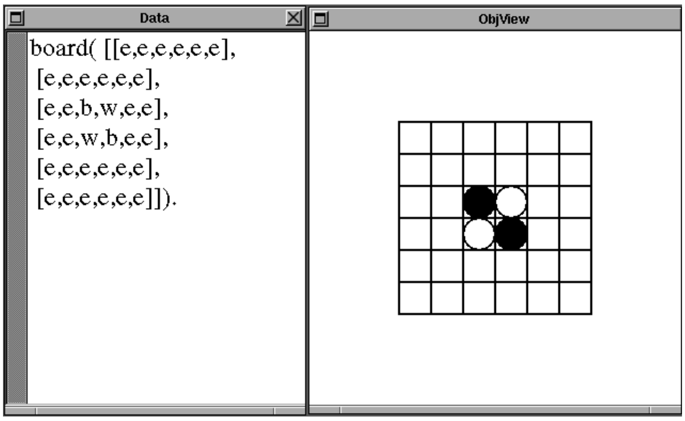
The objective of declarative specification is also achieved with CONSTRAINT[34] with a schema called *constraint grammar*. The major difference with TRIP2, however, is that the data in an application object are accessed directly, rather than to provide the layers of abstractions as is with TRIP2. Therefore, an application and its interface become strongly interdependent with CONSTRAINT, making interface specification and modification more complex compared to TRIP2.

6 Concluding Remarks

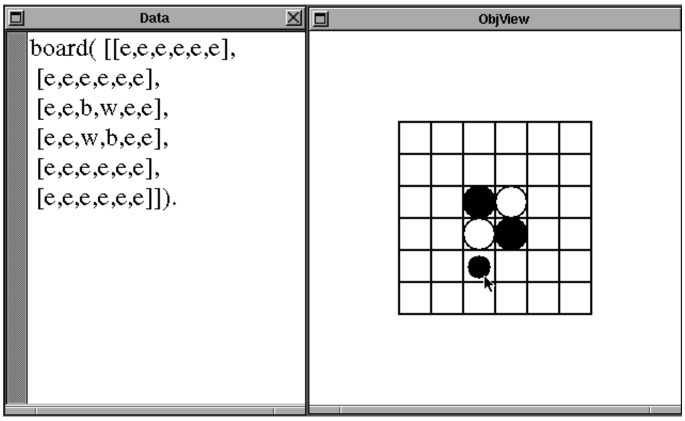
We have presented an overview of how direct manipulation can be achieved in our proposed bi-directional translation model, and have described its prototype implementation, TRIP2, and the examples of its use. In all the examples, by specifying only the declarative rules, the users can directly modify a picture, and the manipulations are fed back to abstract structure data automatically by the system. We stress that, since the system-provided user interface is independent of ASR data in TRIP2, (1) the system is applicable to a variety of types of ASR data, and (2) different visualization and its DM interfaces can be specified for a single type of ASR data, by merely providing the appropriate mapping rules. Thus, we have made substantial progress in achieving *dialogue independence* in the context of DM-style interfaces, a task which was pointed out as being significantly difficult in [8].

Our experiences with TRIP2 also made us aware of some new research issues. First is to improve the turnaround time of feedback. For the example of kinship diagram, the turnaround time is about 10 seconds, and for the Othello game, the turnaround time is about 18 seconds. Most of the time is spent for the execution of Prolog. We are tackling this in several ways:

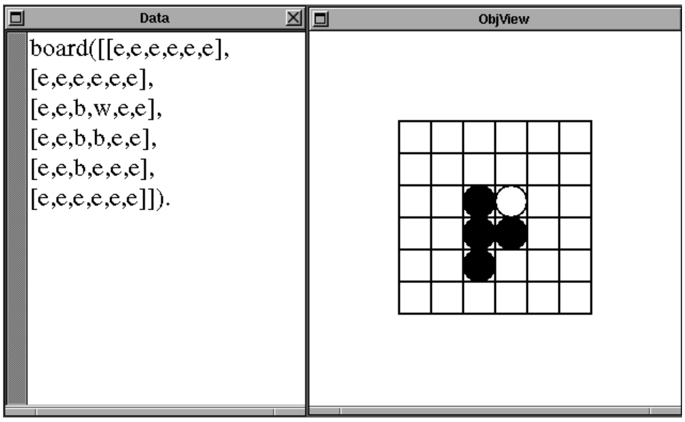
[4] It is of course possible and useful to apply TRIP2 to visualization of numerical data.



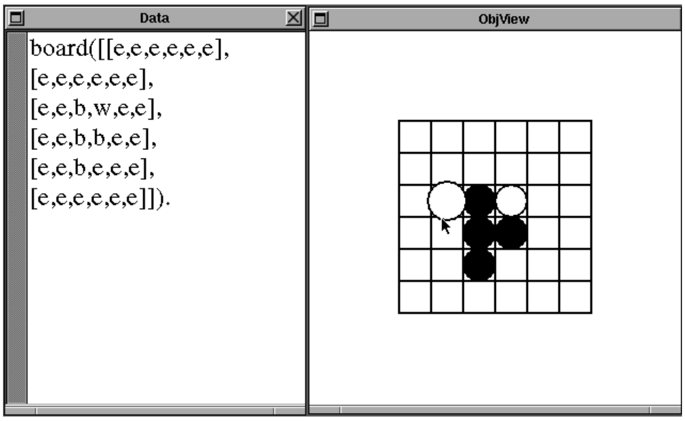
(a) The board-data in the left window are mapped to the picture in the right window.



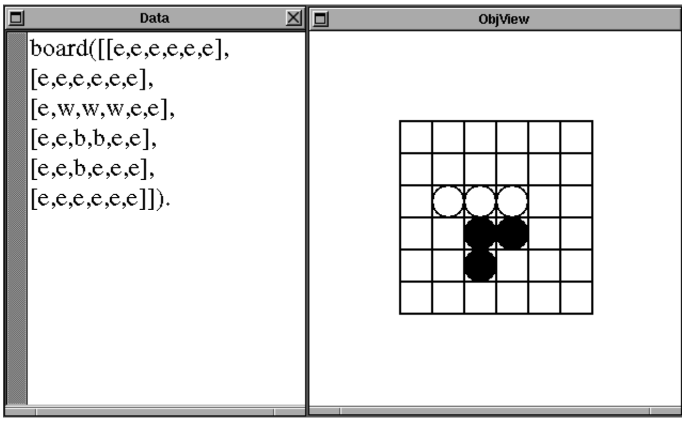
(b) The user draws a black circle. (note that certain margins are allowed for the size and the position of the piece the user draws.)



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



(d) The user draws a white circle.



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

Figure 6: TRIP2 Othello

- *Employing faster Prolog or other symbolic manipulation languages:* The use of Prolog as a symbolic constraint-solving language facilitated fast, reliable construction of our prototype. However, Prolog is not necessarily well-suited for our task with respect to efficiency of execution. By employing other symbolic manipulation languages suited for our task, we should improve the speed of visual mapping and inverse visual mapping substantially. (See discussion on constraint logic programming language later on in this section.)
- *Decreasing the interlingual translation overhead:* The current prototype employs a naive scheme for interlingual interface between Prolog and Objective-C: data are passed to/from the Prolog module from/to the Objective-C module via a Unix file with VSR objects and relations encoded in intermediate textual form. This is necessary because the Prolog interpreter does not have the capability to access Objective-C objects and vice-versa[5]. Our benchmarks have shown that this low-level translation process is a considerable bottleneck, and we are in the process of combining these modules in an integral manner.
- *Incremental translation:* At present, the translation process is *total* at each stage (except for spatial parsing), meaning that the entire data in each representation is translated into the next representation. Since changes are usually limited to some local part of the application data or the picture, incremental translation at each stage would substantially decrease the amount of computation required.

The latter two strategies are being tested with the DeltaTRIP system[26] implemented and running on ParcPlace Smalltalk v2.5. The translation in DeltaTRIP is one-way from application data into pictures. The differences with TRIP are that (1) the Prolog interpreter is now an integral part of the system, being able to manipulate Smalltalk objects, and (2) the translation process at each stage is incremental as discussed above. (The underlying incremental constraint solver is DeltaBlue[6], courtesy of John Maloney). Compared to TRIP2, preliminary tests have shown substantial increase in the speed of visualization when the ASR data to be visualized is modified incrementally (each incremental translation process takes less than a second). The speed improvement achieved in DeltaTRIP is very promising, because the Prolog interpreter used in DeltaTRIP, being written in Smalltalk, is even an order of magnitudes slower compared to the one employed in TRIP2.

Secondly, as indicated earlier, the users have to write both the rules for a visual mapping and the corresponding inverse visual mapping, although they are almost the same in structure (see AppendixB for mapping rules of the graph editor example in Section 4.1). We are working to either (1) improve the system so that a single rule would truly serve both purposes, or (2) devise some scheme for automatically deriving one from the other.

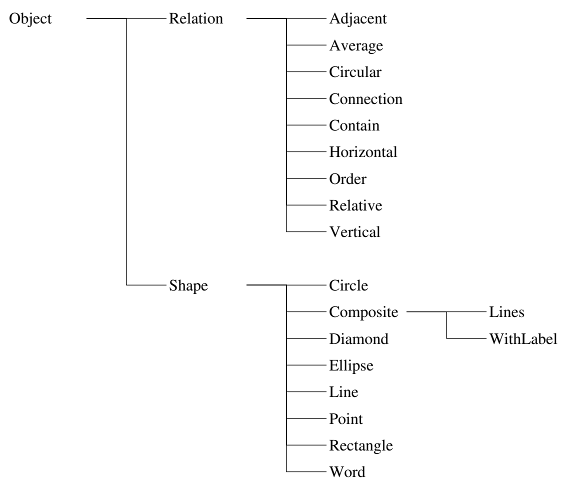
A more fundamental issue is that writing mapping rules in Prolog requires some practices and experiences for novice users, although much simpler than writing procedural programs. Still, to make it easier for users to write rules, a friendlier language for describing mapping rules is desired. Moreover, specifying mapping rules visually via user demonstration would be more attractive than writing rules in a textual language. These are our undergoing research topics.[6]

Another way of extending TRIP2 is to use a constraint (logic) programming language, such as PROLOG III[4] and CLP(R)[9]. The numerical constraint solver in TRIP2 is external to the Prolog in the current implementation. As a result, it is difficult to write mapping rules which the high-level mapping rules when constraints can not be solved in the lower translation stages. By integrating the constraint solving stages into logic programming with a constraint programming language, such mapping rules would be possible, allowing specification of more elaborate visualization. Such a language would also be beneficial for the inverse translation.

Acknowledgments

We would like to thank Professor Satoru Kawai who gave us valuable comments in the early stage of this research. We are also grateful to Professor Masami Hagiya and the members of TRIP group for the helpful discussions, and the members of Yonezawa Lab. for their encouragement. Finally, we thank the anonymous referees, whose comments were very valuable in revising this paper.

A Classes in TRIP2



A class for a graphical relation is derived from the *Relation* class, and a class for a picture object is derived from the *Shape* class. Picture objects (in PR) are used also as graphical objects (in VSR) in the interaction module.

B A Mapping Rule Example

```
%% Mapping Rules for Graph Editor
%   Sample data
%       edge(a,b).      edge(b,c).
%       edge(a,c).      edge(a,d).
%       node(a).        node(b).
%       node(c).        node(d).
```

[5] As a matter of fact, integration can significantly increase the speed in practice: TRIP2 consists of several submodules, and the data transfers between the Objective-C modules, such as constraint solver, and interaction module, are also done by using text files in the current implementation.

[6] We are currently investigating Muggleton and Buntine's *inverting resolution* in CIGOL[18] as an underlying framework for generating the mapping rules automatically from user demonstration.

```
% Visual mapping driver
	objectmap([o]).
	o :- node(X), nodemap(X), fail.
	relationmap([r]).
	r :- edge(A, B), edgemap(A, B), fail.

% Visual mapping rules
	nodemap(X) :-
		circle(X, 15, []),
		label(X, X, []).
	edgemap(A, B) :-
		connect(A, B, center, center, []),
		adjacent(A, B, 1).

% Inverse visual mapping driver
	invomap([io]).
	invrmap([ir]).
	io :-
		invnodemap(X),
		write('node('), write(X), write(').'), nl,
		fail.
	ir :-
		invedgemap(A, B),
		write('edge('), write(A), write(','),
		write(B), write(').'), nl,
		fail.

% Inverse visual mapping rules
	invnodemap(X) :-
		circle(X, _, _),
		label(_, X, _).
	invedgemap(A, X) :-
		connect(A, X, _, _, _).
```

References

[1] Apple Computer, Inc., *Inside Macintosh*, 1985.

[2] Borning, A. and R. Duisberg, "Constraint-Based Tools for Building User Interfaces," *ACM Trans. Graphics*, vol. 5, no. 4, pp. 345–374, October 1986.

[3] Chang, S. K., *Principles of Visual Programming Systems*. New Jersey: Prentice-Hall, 1990.

[4] Colmerauer, A., "An Introduction to Prolog III," *Communications ACM*, vol. 33, no. 7, pp. 69–90, July 1990.

[5] David L. Maulsby, I. H. W. and K. A. Kittlitz, "Metamouse: Specifying Graphical Procedures by Example," *Computer Graphics*, vol. 23, no. 3, pp. 127–136, July 1989.

[6] Freeman-Benson, B. N., J. Maloney, and A. Borning, "An Incremental Constraint Solver," *Communications ACM*, vol. 33, no. 1, pp. 54–63, Jan 1990.

[7] Golin, E. J. and S. Reiss, "The Specification of Visual Language Syntax," in *IEEE Workshop on Visual Languages*, pp. 105–110, 1989.

[8] Hartson, H. R. and D. Hix, "Human-Computer Interface Development: Concepts and Systems for Its Management," *ACM Computing Surveys*, vol. 21, no. 1, pp. 5–92, March 1989.

[9] Jaffar, J. and J.-L. Lassez, "Constraint Logic Programming," in *Fourteenth ACM Symposium on the Principle of Programming Languages*, 1987.

[10] Kamada, T., *Visualizing Abstract Objects and Relations, A Constraint-Based Approach*. Singapore: World Scientific, 1989.

[11] Kamada, T. and S. Kawai, "An Algorithm for Drawing General Undirected Graphs," *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, April 1989.

[12] Kamada, T. and S. Kawai, "A General Framework for Visualizing Abstract Objects and Relations," *ACM Trans. Graphics*, vol. 10, no. 1, pp. 1–39, Jan. 1991.

[13] Knuth, D. E., "Computer-Drawn Flowcharts," *Communications ACM*, vol. 6, no. 9, pp. 555–563, September 1963.

[14] Lakin, F., "Visual Grammers for Visual Languages," in *Proceedings of AAAI87, Sixth National Conference on Artificial Intelligence*, July 1987.

[15] Lakin, F. H., "A Structure from Manipulation for Text-Graphic Objects," in *Proceedings of SIGGRAPH '80*, July 1980.

[16] Mackinlay, J., "Automating the Design of Graphical Presentations of Relational Information," *ACM Trans. Graphics*, vol. 5, no. 2, pp. 110–141, April 1986.

[17] Maloney, J., A. Borning, and B. N. Freeman-Benson, "Constraint Technology for User-Interface in ThingLabII," in *OOPSLA '89 Conference Proceedings*, pp. 381–388, 1989.

[18] Muggleton, S. and W. Buntine, "Machine Invention of First-order Predicates by Inverting Resolution," in *Proceedings of Machine-Learning*, (Michigan), Morgan Kaufmann, June 1988.

[19] Myers, B. A., "Creating Highly-Interactive and Graphical User Interfaces by Demonstration," *Computer Graphics*, pp. 249–258, 1986.

[20] Myers, B. A. et al., "Garnet, Comprehensive Support for Graphical, Highly Interactive User Interfaces," *IEEE computer*, pp. 71–85, November 1990.

[21] Nakayama, K., S. Matsuoka, and S. Kawai, "Visualization of Abstract Concepts Using Generalized Path Binding," in *Proceedings of CG International '90*, pp. 377–402, Springer-Verlag, 1990.

[22] Nassi, I. and B. Shneiderman, "Flowchart Techniques for Structured Programming," *ACM SIGPLAN*, pp. 12–26, August 1973.

[23] Nelson, G., "Juno, a Constraint-Based Graphics System," *Computer Graphics*, vol. 19, no. 3, pp. 235–243, July 1985.

[24] NeXT Inc., *NeXT System Reference Manual*.

[25] Pineda, L. A. et al., "GRAFLOG: Programming with Interactive Graphics and PROLOG," in *Proceedings of CG International'88*, pp. 469–478, Springer-Verlag, 1988.

[26] Sakai, N., "Visualization of Abstract Data," Master's thesis, Tokyo Institute of Technology, 1990. (In Japanese).

[27] Shneiderman, B., "Direct Manipulation: A Step Beyond Programming Languages," *IEEE computer*, pp. 57–69, August 1983.

[28] Shu, N. C., *Visual Programming*. Van Nostrand Reinhold, 1988.

[29] Stanford University, *InterViews Reference Manual*.

[30] Teitelman, W. and L. Masinter, "The Interlisp Programming Environment," *IEEE Computer*, vol. 14, no. 4, pp. 25–34, April 1981.

[31] Vlissides, J. M. and M. A. Linton, "Unidraw: A Framework for Building Domain-Specific Graphical Editors," in *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 158–167, 1989.

[32] Wolfram, S., *Mathematica, A System for Doing Mathematics by Computer*. Addison-Wesley, 1988.

[33] Wong, H. K. T. and I. Kuo, "GUIDE: A Graphical User Interface for Database Exploration," in *Proceedings of the Conference on Very Large Databases*, pp. 22–32, 1982.

[34] Zanden, B. T. V., *Incremental Constraint Satisfaction And Its Application To Graphical Interfaces*. PhD thesis, Department of Computer Science, Cornell University, October 1988.