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

Satoshi Matsuoka Shin Takahashi Tomihisa Kamada*

Akinori Yonezawa†

Department of Information Science, the University of Tokyo

7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 Japan

Research and Development, ACCESS CO., LTD.

1-7-1 Sarugaku-cho, Chiyoda-ku, Tokyo, 101 Japan

Abstract

The merits of direct manipulation are now widely recognized. However, direct manipulation interfaces incur high cost in their creation. To cope with this problem, we present a model of bi-directional translation between pictures and abstract application data, and 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 easy creation of direct manipulation interfaces. We apply the prototype system to the generation of the interfaces for kinship diagrams, Graph Editors, E-R diagrams, and an Othello game.

Categories and Subject Descriptors: D.2.2 [Software Engineering]:Tools and Techniques–user interfaces; H.1.2 [Models and Principles]:User/Machine Systems–human information processing

General Terms: Algorithms, Design, Experimentation, Human Factors

Additional Keywords: Bi-Directional Translation, Direct Manipulation, User Interface, User Interface Management Systems, Visualization

*Email: tomy@access.co.jp

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

To appear in ACM Transaction on Information Systems

1 Introduction

Graphical and interactive user interfaces based on *direct manipulation*[30], allowing the users to point to, grasp, and drag objects on the screen, are now standard. To ease the high creation cost, we employ User Interface Management Systems (UIMSs) and user interface toolkits such as NextStep[26], MacApp[1] and InterViews[32]. However, development of direct manipulation style interfaces is still costly, because UIMSs and toolkits usually only provide 'canned' abstractions of behaviors for a fixed set of simple graphical interface objects such as buttons and scrollbars. Such a fixed set of abstractions may be sufficient for simple, dialogue-style interfaces, but for cases where the semantics of complex application data must be reflected and be manipulatable on the display, appropriate graphical and input abstractions may not always be available. This is primarily because current models of user interfaces are usually the models of the interaction architecture, and lack support for consistent frameworks that allow visualization and manipulation of high-level abstract data, i.e., semantics of applications.

In our previous research, TRIP[12, 14] 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, application data is represented by abstract objects and relations, which are mapped to graphical objects (such as boxes and circles) and graphical relations (such as horizontal/vertical listings and line connections). The mapping is specified not with procedures, but with declarative mapping rules written in Prolog, a logic programming language.

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 with little 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, however, is in its applicability as a general-purpose schema. Since there are infinite kinds of both application's data representations and its pictorial representations, ad-hoc schemes would force the users to specify complex representation-dependent translation rules for every new combinations of abstract and picture data.

To solve 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 application data or pictorial representations change, the mapping rules between ASR and VSR are essentially unaffected.

Altogether, the translation process handles the following 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 of application data, which is a set of *abstract 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 data representation of the application is generated from this representation. In TRIP2, ASR data are represented as a set of Prolog predicates defined by the programmers.

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. In TRIP2, VSR data are also represented as a set of Prolog predicates. In contrast to ASR predicates, which are user-defined, VSR data are special predicates predefined in the system corresponding to graphical objects and relations.

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

Figure 1 illustrates an example of the four types of representations and their relationships: The application's data is represented as a natural language sentence "x consists of p, q, and r". ASR and VSR data are represented as a set of Prolog predicates[1]. The ASR data corresponding to the above English sentence are consists_of(x, [p, q, r]), and is_object(x), is_object(p), etc., where consists_of() is an abstract relation,

[1]Predicates have the form `pred_name(param1, param2, ...,param_n)`. `Params` could be ground literals such as integers and strings, structures, or variables (capitalized). Square brackets ('[' and ']') are used to delimit lists.

and is_object()s are abstract objects, which are defined by the application programmer. 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[2]. Above(), hor-listing(), and mconnect() are graphical relations, and the boxwithlabel()s are graphical objects. These predicates are predefined in the system, and its parameters represent their attributes. The first parameter of a graphical object is its identifier (Constants, lists, and predicates can be used as an identifier.); graphical relations use these identifiers to designate their constituent graphical objects. As a pictorial representation, a picture of a tree is visualized in Figure 1.

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 [12, 14].

First, the original application's data representation is analyzed, and its ASR data are generated (Figure 1 (1)). Analyzers are customized for each kind of application data, since they are dependent on the particular syntax of application data.

Secondly, ASR data is translated into VSR data in a process we refer to as *visual mapping* (Figure 1 (2)). Abstract objects are mapped to graphical objects, and abstract relations are mapped to graphical relations among the corresponding graphical objects. Users can create a variety of pictures merely by changing the mapping rules. The inverse translation from VSR data into ASR data is referred to as *inverse visual mapping* (Figure 1 (5)). The details of the mapping rules are described in the ensuing sections.

We make a note here that the mapping rules are declarative, and thus do not embody directional information of translation. As a result, a single rule could conceptually be employed both for visual mapping and for inverse visual mapping[3]. By contrast, if we were to specify the mappings procedurally, it would be necessary for us to program two totally distinct procedures for each direction.

Finally, VSR data are translated into target pictorial representation (PR) data (Figure 1 (3)). The graphical relation data are first translated into geometrical constraints among *picture objects*. To determine their actual positions (x- and y-coordinates) in a display space, the constraints are resolved by constraint solvers. Then, device-dependent picture data are generated, which are rendered on the display. In the inverse translation, VSR data are generated from picture data by performing *spatial parsing* on the picture (Figure 1 (4)); the details are described in Section 3.4.

[2] The VSR data here denotes the following pictorial structure: Object x is placed above objects p, q, and r. Objects p, q, and r are horizontally arranged. Object x is connected to objects p, q, r by solid orthogonal lines. Object x is a box with label 'x'.

[3] Our current implementation requires two rules for mappings. See Section 3.4 for more details.

The inverse translation is executed as follows: first, picture data are analyzed with the spatial parser (Figure 1 (4)), and VSR data are generated; second, the VSR data are mapped to ASR data via *inverse visual mapping* (Figure 1 (5)); finally, application data are generated from ASR data with an application specific synthesizer (Figure 1 (6)).

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)[12, 14], 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 users to manipulate the pictures directly with input devices such as a mouse. Currently, this module provides users with an object-oriented freehand drawing interface similar to MacDraw[4]. 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 invocations of Objective-C objects that represent the picture.

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

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

In TRIP2, since visual mapping and inverse visual mapping are executed in Prolog, we also use Prolog's *asserted predicates* (or, *facts*) for ASR and VSR data. The spatial parser is integrated with the interaction module in the current implementation. The details of these modules are described in the following sections.

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.

[4] MacDraw is a trademark of Claris, Inc.

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

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

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

6. Return to 4.

Here we present an example of using TRIP2 in Figure 3: Each screen dump of left-right windows shows the data of a Japanese family as ASR data and its corresponding picture visualized 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 user is free to modify either data at any time (He interacts only with the pictures in the *ObjView Window* in this example.). Now, suppose that a new member, *Midori*, joins the family by marrying *Shinichi*. To express this fact, we draw a rectangle (Figure 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 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 again. These modifications are fed back, and the diagram is regenerated (Figure 3(g)).

3.2 The TRIP Module

The TRIP[12, 14] 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) the layout system called COOL (COnstraint-based Object Layout system).

The analyzer translates application's data into ASR data, which is a set of Prolog predicates. In [14], 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 parts of TRIP2; rather, TRIP2 provides an editing window for ASR data as shown in Figure 3[5]. Thus, the users can interactively

[5] In fact, the original TRIP system was interfaced with a simple natural language parser; as a result, the user could input sentences describing a scenery, and TRIP would automatically visualize it[14]. This parser can easily be interfaced with TRIP2 as well.

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 mapping rules are read into the database, after which the visual mapper is invoked.

Finally, VSR data are translated into PR data by COOL. In COOL, VSR data are declared as special predicates that generate (a) drawing data, and (b) constraints on the geometric attributes in the drawing data. For example, predicate `box` generates (a) data for drawing a rectangle, and (b) constraints on coordinate variables of the rectangle. The predicate `horizontal` generates (b) constraints on y-coordinate variables of picture objects corresponding to the graphical objects listed in its argument. These constraints are solved by the constraint solvers that are parts of COOL. The constraint solvers can handle two types of constraints: 'rigid' constraints, which must be solved exactly, and 'pliable' constraints, which may be solved approximately by the least square method. The purpose of pliable constraints is to allow COOL to handle over-constrained systems. In addition, COOL has special predicates and its associated constraint solver for visualizing undirected graph structures as balanced graphs. The constraint solving algorithms of COOL are described in [12, 14]. The solutions computed by the constraint solvers instantiate the coordinate variables in the drawing data. They are then passed to the interaction module to generate the Objective-C picture objects.

For example, given the ASR data:

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

and the visual mapping rules (Figure 4), the VSR data in Figure 5 are generated.

To execute the visual mapping, TRIP calls the predicates listed in the argument of predicates `objectmap` and `relationmap`. The argument predicates of `objectmap` (`itemmapping` in the example shown in Figure 4) map abstract objects to graphical objects, and the argument predicates of `relationmap` (`ordermapping` in Figure 4) map abstract relations to graphical relations. When the predicate `itemmapping` is called by the system (Figure 6 (1)), the subgoal `item(NAME)` matches a single instance of ASR data in the database (`item('AR')`), and variable `NAME`[6] is instantiated to `'AR'` via pattern matching and variable instantiation (i.e., unification) (Figure 6 (2)). Then, subgoal `itemmap(NAME)` matches with the rule `itemmap(NAME)` (Figure 6 (3)), and three VSR predicates are output as VSR data, which are `box()`, `label()`, and `contain()` (Figure 6 (4)). This process can essentially be regarded as procedure calls (i.e., as if the procedure `itemmap` calls the procedures `box`, `label`, and `contain`.). The special predicate `fail` forces repetition of this pro-

[6] In Prolog, any name beginning with a capital letter is taken to be a variable.

cess with the other `item`s via backtracking until they are all mapped to VSR — that is, `itemmapping` is an iteration which maps all `item`s to VSR. Similarly, the predicate `ordermapping` applies the predicate `ordermap` to the abstract relation `order(ILIST)`, and it maps `order(ILIST)` to a graphical relation `horizontallisting`. This results in the '`item`'s being layed out horizontally. Next, COOL is invoked and the PR shown in Figure 7 is generated from the VSR data.

3.3 The Interaction Module

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, users simply draw rectangles or circles as if they were using object-oriented freehand drawing systems such as MacDraw. This is unlike graphical editors which are customized for each application, such as Unidraw[35].

A picture in the interaction module is represented as a set of picture objects in Objective-C. Picture objects are instances of subclasses of the `Shape` class (see Figure 8), in which the usual generic methods, such as drawing, moving, changing the size/place of the object, reading 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 earlier, 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 the VSR data. 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 VSR graphical objects. Graphical relations in VSR are represented as a list of Objective-C graphical relation objects, which are instances of the subclasses of the `Relation` class (see Figure 8), which supports the following methods:

- A method for calculating the degree of deviation of the constituents 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 the translation from PR into VSR as *spatial parsing*. Although spatial parsing has been previously studied in general contexts (for example, [9, 16, 17]), its application to GUI poses some 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; its purpose is to extract a set of graphical relations among graphical objects. Since pictorial objects are recognized as graphical objects in the interaction module, the spatial parser in TRIP2 needs only to extract the graphical relations.

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

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

- **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, when a new rectangle is drawn with a label `e` to the right of rectangle `d` (Figure 9), `addWithTest:e` message is sent to the graphical relation object, `x_order([a,b,c,d],10)`. In this case, the distance `w` between `d` and `e` is nearly equal to 10; as a result, the new rectangle object is added to the `x_order` relation[7]

- **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

[7] In this manner, the spatial parser permits a certain amount of tolerance.

cannot handle the creation of new graphical relations. For example, the `connect` relation, which represents two graphical objects connected by a line, must be created when a user draws a line that connects two picture objects that were previously unrelated. We alleviate this by associating generations of new graphical relations only with drawings of the picture objects — that is, when a certain kind of an object is drawn, the system checks whether the object has established new relations or not. For example, as shown in Figure 10, when a line connecting two objects is drawn, a `connect` relation is created; and when an object is drawn within the boundary of another object, a `contain` relation is created. We are now working to provide alternative means of generating other types of graphical relations.

In TRIP2, all graphical relations are inferred by the spatial parser. One problem of this method is that users cannot immediately judge whether the system is inferring his intentions correctly or not. It is thus preferable that inferred constraints are visually displayed. In addition, there may be cases where explicit selection of constraints from a given set of inferred constraints is desired. These problems have been addressed in our latest TRIP3 system, a successor to TRIP2. (See Section 6.)

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

Ideally, we should be able to use identical rules for inverse visual mapping as well as for visual mapping, because declarative rules do not have 'directional' information. However, this is not the case due to the following reason: If some rule is used naively for mappings in both directions, the pictorial editing performed by the user must exactly satisfy the layout constraints imposed by the rule. The resulting interface would be extremely rigid and difficult to use — for example, in Figure 4, the user would be required to draw a rectangle of the exact size as specified by the mapping rule ($width = 200, height = 100$) for his input to be recognized.

Instead, we must relax the constraints so that system would be tolerant of the non well-formed operations of the user. In TRIP2, this is currently achieved by having the user provide a separate set of corresponding inverse visual mapping rules which would be 'less specific' compared to its visual mapping counterparts. For example, in Figure 11, the predicate `box` has several of its variables unspecified (this is indicated by underscores ('_'), which serves as *anonymous variables* in Prolog). As a result, the height and the width of the user-drawn box are ignored, allowing the user to draw a box of arbitrary size[8].

[8] This is one of the reasons why we are able to first do the inverse mapping, and then perform the visualization again to achieve beautification; the beautified PR is in effect the 'exact' visualization of a given ASR data, whereas the edited picture

Fortunately, the structure of the mapping rules are often nearly identical; thus, an experienced user will be able to derive one from the other with ease. Moreover, automatic derivation of inverse mapping rules from (forward) mapping rules could be possible. (See Section 6.)

For example, the inverse visual mapping rules that correspond to the visual mapping rules in Figure 4 are shown in Figure 11. They are executed as follows: First, `inv_itemmapping` is called (Figure 12 (1)). Second, the subgoal `inv_itemmap(NAME)` calls the head of the rule `inv_itemmap(NAME)` via unification (Figure 12 (2)). At this point, the variable `NAME` is still unbounded. Then the three predicates (`box`, `label`, and `contain`) are unified (matched) with a set of VSR data in the database, and the variables `A`, `B`, and `NAME` are instantiated to `oid1`, `oid2`, and `'AR'`, respectively (Figure 12 (3)). Here, the predicate `box`, `label`, and `contain` in `inv_itemmapping` can be regarded as a template for searching a pair of VSR data corresponding to the ASR `item`. Thus, `NAME` in subgoal `inv_itemmapping` is bound to `'AR'` (Figure 12 (4)), and the predicate `asr` generates the ASR data `item('AR')` (Figure 12 (5)). Since the predicate `fail` forces backtracking, the above process is repeatedly applied to the other VSR data, i.e., `inv_itemmapping` can be regarded as an iteration that searches and generates all the `item`s. Similarly, `order(ILIST)` is obtained from the `horizontallisting` relation in the VSR database, and output in a certain format. Compared to the visual mapping rules (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 is similar.

Often, diagram interpretation involves multiplicity of interpretations. In TRIP2, since a diagram is interpreted with inverse mapping rules, the problem of multiple interpretations can be solved by writing the appropriate rules. When the mapping rule allows multiple interpretations, the ASR data of all the interpretations are usually generated, because inverse mapping rules search with backtracking for all possible applications of the rules as is with the rules in Figure 11.

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

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

just prior to beautification is not.

4 Examples

In this section, we show examples of TRIP2 in action. All the figures in this section show the screen dumps of two windows of TRIP2. Specifically, 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. Appendix A.1 gives the visual mapping rules for this example: there, nodes are mapped to circles, and edges are represented as straight lines. Users of this editor simply draw circles, lines, and labels. These pictures are recognized as a graph, and translated into `node`s and `edge`s. The nodes in this example are arranged by the `adjacent` relation, which is an entry to the graph layout constraint solver in TRIP2. The graph drawing algorithm used here is described in [12, 13].

4.2 A Simple E-R Diagram Editor

Figure 14 shows the screen snapshots of the windows of an Entity-Relationship Diagram editor. The upper-left window shows the E-R database schema in a textual form, and the right window displays its E-R diagram. The user extends the schema by adding a new entity *course* (Figure 14(b)) with two attributes (*number* and *name*) (Figure 14(d)), and new relationships (*S-C* and *I-C*) (Figure 14(c)). The user then performs the *flush* command. The modified picture in the right window is translated back to the E-R schema data in ASR, and these data are translated into the corresponding beautified E-R diagram (Figure 14(e)).

In this example, entities (represented by rectangles), and attributes (represented by ellipses) are arranged by the graph layout constraint solver as is with the previous graph editor example. The mapping rules for this editor are described in Appendix A.2.

4.3 A Small Othello Game

This example illustrates an interface for a small Othello game application. Figure 15 shows the screen dumps of the windows during the game, which display the board data (ASR) and its pictorial representation. The Othello application itself is written in Prolog, and uses this ASR data directly as its representation of the board (therefore, there is no translation process between AR and ASR). It has no knowledge of the interface of the game. The board data consists of six rows of six characters (Figure 15(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 15(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 15(c)). Player-2 then draws a white stone (Figure 15(d)). This modification is also fed back to the board data (Figure 15(e)).

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*[36] are now in widespread use. Business graph packages are also used extensively in office and personal work. One interesting work is by Mackinlay[18], 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[9], 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 [15, 24, 33], and visualization of database schema [37] has been studied. Here, the advantage of TRIP2 (and TRIP) is that it is a general-purpose system that can be applied to numerous types of visualization simply by defining declarative mapping rules. Declarative approach to program visualization is proposed by Cox and Roman[28, 5] in the system *Pavane*. Pavane provides three-dimensional animation of concurrent program written in the *Swarm* notation via specification of declarative rules. The difference from TRIP2 is that (1) it is able to handle temporal relations for animation, but (2) it does not use constraint satisfaction techniques for layouts, and (3) it does not support inverse mapping from pictures to programs. In the context of TRIP, a general framework for visualization of abstract data and relations that can be mapped to ordering relations among the constituent objects is presented in [23].

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[21], and Interface Builder[26]. *Visual programming* systems also recognize user's operations, but are special-purpose in the sense that

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

the interfaces are usually tailored for a single (visual) language; they are surveyed in [3, 31]. Of special interest are GRAFLOG[27] which is an interactive graphics interface in which drawings receive linguistic interpretations, i.e., a certain kind of visual language, PAM[17] which is a system for manipulating text-graphic patterns, and Golin's Palette system[8], which is a tool for constructing visual program editors using the Indigo language, which is a visual language for controlling a visual program editor. Constraint-based graphical UI systems such as Juno[25], ThingLab[2], ThingLabII[19], Metamouse[6] 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[22] and Unidraw[35] 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[38] 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 Creating Mapping Rules via Demonstration

For creating direct manipulation interfaces with TRIP2, programmers have only to write visual and inverse visual mapping rules. However, TRIP2 provides little support for writing mapping rules — that is, programmers must write pairs of mapping rules in Prolog. This is not entirely ideal for the following reasons:

- It is cumbersome that 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. We are now almost successful in automatically deriving one from the other.

- A more fundamental issue is that writing mapping rules in Prolog, although much simpler than writing procedural programs, requires some practices and experiences for novice users. A friendlier language for describing the mapping rules is thus desired.

Specifying mapping rules visually via graphical user demonstration, rather than writing rules in a textual language, would greatly reduce the difficulty of writing the rules. To realize this ideal, our group has developed TRIP3, which generates mapping rules automatically from a set of given example pictures and the corresponding example ASR data. We call this style of specification *programming by visual example*. By using TRIP3, mapping rules for various abstract application data such as S-expressions and tree structures can be automatically generated.

Let us follow the process of generating a mapping rule for the organization diagram in Figure 16. First, while drawing a sample visualization, every time the designer draws a new object, the TRIP3 system infers the geometrical relationships such as `horizontal` that are 'likely' to match the designer's intention, and displays the inference to the designer in two ways: (1) In the pict editor view, colored dotted lines indicate the inferred geometrical relationships — in Figure 17, for example, dotted horizontal lines indicate that graphical objects along the lines are aligned horizontally. (2) In the confirmation panel (Figure 18), the system displays its inference explicitly as textual annotations. The designer then selects his intended geometrical relationships with checkboxes in the confirmation panel.

Next, the designer inputs the corresponding application data — Figure 19 depicts the corresponding application data for the organization diagram. The system then generalizes these two types of sample data, and generates a mapping rule reflecting the designer's intention about the layout. The detailed description on the generalization mechanism will be the subject of our next paper[20].

7 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, 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 direct manipulation 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 direct manipulation style interfaces, a task which was pointed out as being significantly difficult in [10].

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 3 seconds, and for the Othello game, the turnaround time is about 5 seconds. Most of the time is spent for the execution of Prolog. We are tackling this in several ways:

- *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.

- *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 using Unix sockets and pipes with VSR objects and relations encoded in an intermediate textual form. This is necessary because the Prolog interpreter does not have the capability to access Objective-C objects and vice-versa[10]. Our benchmarks have shown that this low-level translation process is still a major bottleneck, although we have improved the translation speed nearly five-fold by re-designing the interlingual interface. We are in the process of combining these modules in a more 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 last strategy is being tested with the DeltaTRIP system[29] implemented and running on ParcPlace Smalltalk v2.5 (Figure 20), although it does not support inverse translation from pictures to application data. The differences from TRIP2 are that (1) the Prolog interpreter is now an integral part of the system, being able to manipulate Smalltalk objects, and (2) the translation process at each stage is incremental as discussed above. (The underlying incremental constraint solver is DeltaBlue[7], 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 are 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.

[10] As a matter of fact, most of the translation time were spent for communication between the submodules: TRIP2 consists of several submodules in the old implementation, and data transfers between the Objective-C module (interaction module,

The second issue is the real-time/semantic feedback problem. Current implementation does not constrain any actions of the users on the editor. However, it is necessary that the system can restrict some users' actions, or constrain the behavior of certain parts of a diagram. In addition, these restrictions may change depending on the state of an application. Because our bi-directional translation model achieves total dialog independence, placing the restriction depending on the state of an application will require extension of our framework. A simple solution is to restrict the behavior of a diagram by using constraints of graphical relations[11]. Another approach is as follows: since the inverse mapping could interpret semantically illegal but syntactically legal actions (such as illegal placement of an Othello stone), the application itself would be able to detect such actions, and cancel it by not changing its data. We believe that for many applications, these strategies would suffice.

Another way of extending TRIP2 is to use a constraint (logic) programming language, such as PROLOG III[4] and CLP(R)[11]. The numerical constraint solver in TRIP2 is external to the Prolog in the current implementation. As a result, it is difficult to write mapping rules which the high-level mapping rules are re-applied when constraints can not be solved in the lower translation stages. By integrating the constraint solving stages into logic programming with constraints, such mapping rules would be possible, allowing specification of more elaborate visualization. Such a language would also be beneficial for the inverse translation.

It is also important to integrate TRIP2 with existing UIMS or toolkits. Although TRIP2 has realized easy creation of direct manipulation interfaces, there are many UIMS functions that TRIP2 does not yet support. For example, TRIP2 does not support easy incorporation of customized widgets; for supporting more complex user interactions, such widgets should be seamlessly integratable into the system. In this context, we are investigating a demonstrational way of specifying widgets from primitive geometric elements[34].

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 the TRIP group (esp. Ken Miyashita) for the helpful discussions, and the members of Yonezawa Lab. for their encouragement. Special thanks to Dr. John Maloney and Professor Alan Borning for providing us with the DeltaBlue

constraint solver, and spatial parser) and Prolog modules (visual and inverse visual mapping module) were done by using text files in the old implementation.

[11] In TRIP2, constraints generated by graphical relations are used only during the layout of a target diagram. The resulting diagram is not constrained by those graphical relations.

constraint solver.

Appendix A A Mapping Rule Example

Appendix A.1 A Small Graph Editor

```
%% Mapping Rules for a Small Graph Editor
% Sample data
%   edge(a,b).     edge(b,c).
%   edge(a,c).     edge(a,d).
%   node(a).       node(b).
%   node(c).       node(d).
% Visual mapping driver
    objectmap([o]).
    o :- node(X), nodemap(X), fail.       % iteration that maps all the nodes
    relationmap([r]).
    r :- edge(A, B), edgemap(A, B), fail.% iteration that maps all the edges
% Visual mapping rules
    nodemap(X) :-     % a node is mapped to a box which contains a label
        circle(X, 15, []), label(l(X), X, []), contain(X, l(X), 0, []).
    edgemap(A, B) :- % an edge is mapped to a connection between specified nodes
        connect(A, B, center, center, []), adjacent(A, B, 1).
% Inverse visual mapping driver
    invomap([io]).
    invrmap([ir]).
    io :- % iteration that searches and outputs all the nodes
      invnodemap(X), asr(node(X)), fail.
    ir :- % iteration that searches and outputs all the edges
      invedgemap(A, B), asr(edge(A, B)), fail.
% Inverse visual mapping rules
    invnodemap(X) :-     % searches for a circle that contains a label
       circle(A, _, _), label(B, X, _), contain(A, B, _, _).
    invedgemap(A, X) :- % searches for a connection, which corresponds to an edge
       connect(A, X, _, _, _).
```

Appendix A.2 A Entity-Relationship Diagram Editor

```
%% Mapping Rules for a Entity-Relationship Diagram Editor
% Sample data
%   entity(institute, [name]).
%   entity(student, [name, grade]).
%   relationship('I-S', institute, n, 1).
% Visual mapping rules
objectmap([entitymap]).  relationmap([relationshipmap]).
entitymap :- % iteration that maps all the entities
    entity(E, A), entitymap(E, A), fail.
relationshipmap :- % iteration that maps all the relationships
    relationship(R, E1, E2, M, N), relationshipmap(R, E1, E2, M, N), fail.
```

```
entitymap(E, A) :- % an entity is mapped to a box which contains
                   % a label, and its attributes are mapped
    box(E, 70, 20, [green]), label(l(E), E, []), contain(E, l(E), 0, []),
    attributemap(E, A).
attributemap(_, []).       % attributes are mapped to an ellipse
attributemap(E, [H|L]) :- % with a label, and are connected to the entity
    ellipse([E|H], 40, 20, [green]), label(l([E|H]), H, []),
    contain([E|H], l([E|H]), 0, []),
    adjacent(E, [E|H], 2), connect(E, [E|H], center, center, [straight,blue]),
    attributemap(E, L).
relationshipmap(R, E1, E2, M, N) :- % a relationship is mapped to a diamond with a label,
                                    % and is connected to its two relating entities
    diamond(d(R), 70, 40, [green]), label(l(d(R)), R, []),
    contain(d(R), l(d(R)), 0, []),
    adjacent(E1, E2, 4), between(d(R), E1, E2, []),
    connectwithlabel(E1, d(R), center, center, [straight,blue], M),
    connectwithlabel(E2, d(R), center, center, [straight,blue], N).
% Inverse visual mapping rules
invomap([ientitymap]).  invrmap([irelationshipmap]).
ientitymap :-        % iteration that searches and outputs all the entities
    ientitymap(E, A), asr(entity(E,A)), fail.
irelationshipmap :- % iteration that searches and outputs all the relationships
    irelationshipmap(R, E1, E2, M, N), asr(relationship(R,E1,E2,M,N)), fail.
ientitymap(E, A) :- % an entity is a box which contains a label
    contain(B, L, _, _), box(B, _, _, _), label(L, E, _),
    iattributemap(B, E, A). % searches attributes of this entity
iattributemap(B, E, A) :- % finds all the attiributes connected to this entity
    findall(L, (con(B, W), iattribute(W, L)), A), !.
iattribute(W, L) :- % an attribute is an ellipse which contains a label
    contain(W, X, _, _), ellipse(W, _, _, _), label(X, L, _).
irelationshipmap(R, EL1, EL2, M, N) :- % a relationship is a diamond with a label
                                       % connected to two enties
    contain(D, DL, _, _), diamond(D, _, _, _), label(DL, R, _),
    conl( E1, D, M), conl( E2, D, N), E1 @< E2,
    contain(E1, L1, _, _), box(E1, _, _, _), label(L1, EL1, _), % an entity
    contain(E2, L2, _, _), box(E2, _, _, _), label(L2, EL2, _). % another entity
con(X, Y) :- connect(X, Y, _, _, _). % ignores order of connection
con(X, Y) :- connect(Y, X, _, _, _).
conl(X, Y, L) :- connectwithlabel(X, Y, _, _, _, L). % ignores order of connection
conl(X, Y, L) :- connectwithlabel(Y, X, _, _, _, L).
```

References

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

[2] Borning, A., and Duisberg, R. Constraint-based tools for building user interfaces. *ACM Trans. Gr. 5*, 4 (October 1986), 345–374.

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

[4] Colmerauer, A. An introduction to Prolog III. *Commun. ACM 33*, 7 (July 1990), 69–90.

[5] Cox, K. C., and Roman, G.-C. Visualizing concurrent computations. In *Proceedings 1991 IEEE Workshop on Visual Languages* (1991), pp. 18–24.

[6] David L. Maulsby, I. H. W., and Kittlitz, K. A. Metamouse: Specifying graphical procedures by example. *Computer Graphics 23*, 3 (July 1989), 127–136.

[7] Freeman-Benson, B. N., Maloney, J., and Borning, A. An incremental constraint solver. *Commun. ACM 33*, 1 (Jan 1990), 54–63.

[8] Golin, E. J. Interaction Diagrams: a visual language for controlling a visual program editor. In *Proceedings 1991 IEEE Workshop on Visual Languages* (1991), pp. 153–158.

[9] Golin, E. J., and Reiss, S. The specification of visual language syntax. In *1989 IEEE Workshop on Visual Languages* (1989), pp. 105–110.

[10] Hartson, H. R., and Hix, D. Human-computer interface development: Concepts and systems for its management. *ACM Comput. Surv. 21*, 1 (March 1989), 5–92.

[11] Jaffar, J., and Lassez, J.-L. Constraint logic programming. In *Fourteenth Annual ACM Symposium on Principle of Programming Languages* (January 1987), pp. 111–119.

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

[13] Kamada, T., and Kawai, S. An algorithm for drawing general undirected graphs. *Inf. Process. Lett. 31*, 1 (April 1989), 7–15.

[14] Kamada, T., and Kawai, S. A general framework for visualizing abstract objects and relations. *ACM Trans. Gr. 10*, 1 (Jan. 1991), 1–39.

[15] Knuth, D. E. Computer-drawn flowcharts. *Commun. ACM 6*, 9 (September 1963), 555–563.

[16] Lakin, F. Visual grammers for visual languages. In *Proceedings of AAAI87, Sixth National Conference on Artificial Intelligence* (July 1987), pp. 683–688.

[17] Lakin, F. H. A structure from manipulation for text-graphic objects. In *Proceedings of SIGGRAPH '80* (July 1980), pp. 100–107.

[18] Mackinlay, J. Automating the design of graphical presentations of relational information. *ACM Trans. Gr. 5*, 2 (April 1986), 110–141.

[19] Maloney, J., Borning, A., and Freeman-Benson, B. N. Constraint technology for user-interface in ThingLabII. In *OOPSLA '89 Conference Proceedings* (1989), pp. 381–388.

[20] Miyashita, K., Matsuoka, S., Takahashi, S., Yonezawa, A., and Kamada, T. Declarative programming of graphical interfaces by visual examples. Submitted for publication.

[21] Myers, B. A. Creating highly-interactive and graphical user interfaces by demonstration. *Computer Graphics* (1986), 249–258.

[22] Myers, B. A., et al. Garnet, comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer* (November 1990), 71–85.

[23] Nakayama, K., Matsuoka, S., and Kawai, S. Visualization of abstract concepts using generalized path binding. In *Proceedings of CG International '90* (1990), Springer-Verlag, pp. 377–402.

[24] Nassi, I., and Shneiderman, B. Flowchart techniques for structured programming. *ACM SIGPLAN* (August 1973), 12–26.

[25] Nelson, G. Juno, a constraint-based graphics system. *Computer Graphics 19*, 3 (July 1985), 235–243.

[26] NeXT Inc. *NeXT System Reference Manual.*

[27] Pineda, L. A., et al. GRAFLOG: Programming with interactive graphics and prolog. In *Proceedings of CG International'88* (1988), Springer-Verlag, pp. 469–478.

[28] Roman, G. C., and Cox, K. C. A declarative approach to visualizing concurrent commputations. *IEEE Computer 22*, 10 (October 1989), 25–36.

[29] Sakai, N. Visualization of abstract data. Master's thesis, Tokyo Institute of Technology, 1990. (In Japanese).

[30] Shneiderman, B. Direct Manipulation: A step beyond programming languages. *IEEE Computer* (August 1983), 57–69.

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

[32] Stanford University. *InterViews Reference Manual.*

[33] Teitelman, W., and Masinter, L. The interlisp programming environment. *IEEE Computer 14*, 4 (April 1981), 25–34.

[34] Tonouchi, T., Nakayama, K., Matsuoka, S., and Kawai, S. Creating visual objects by direct manipulation. In *1992 IEEE Workshop on Visual Languages* (September 1992). To appear.

[35] Vlissides, J. M., and Linton, M. A. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (1989), pp. 158–167.

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

[37] Wong, H. K. T., and Kuo, I. GUIDE: A graphical user interface for database exploration. In *Proceedings of the Conference on Very Large Databases* (1982), pp. 22–32.

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

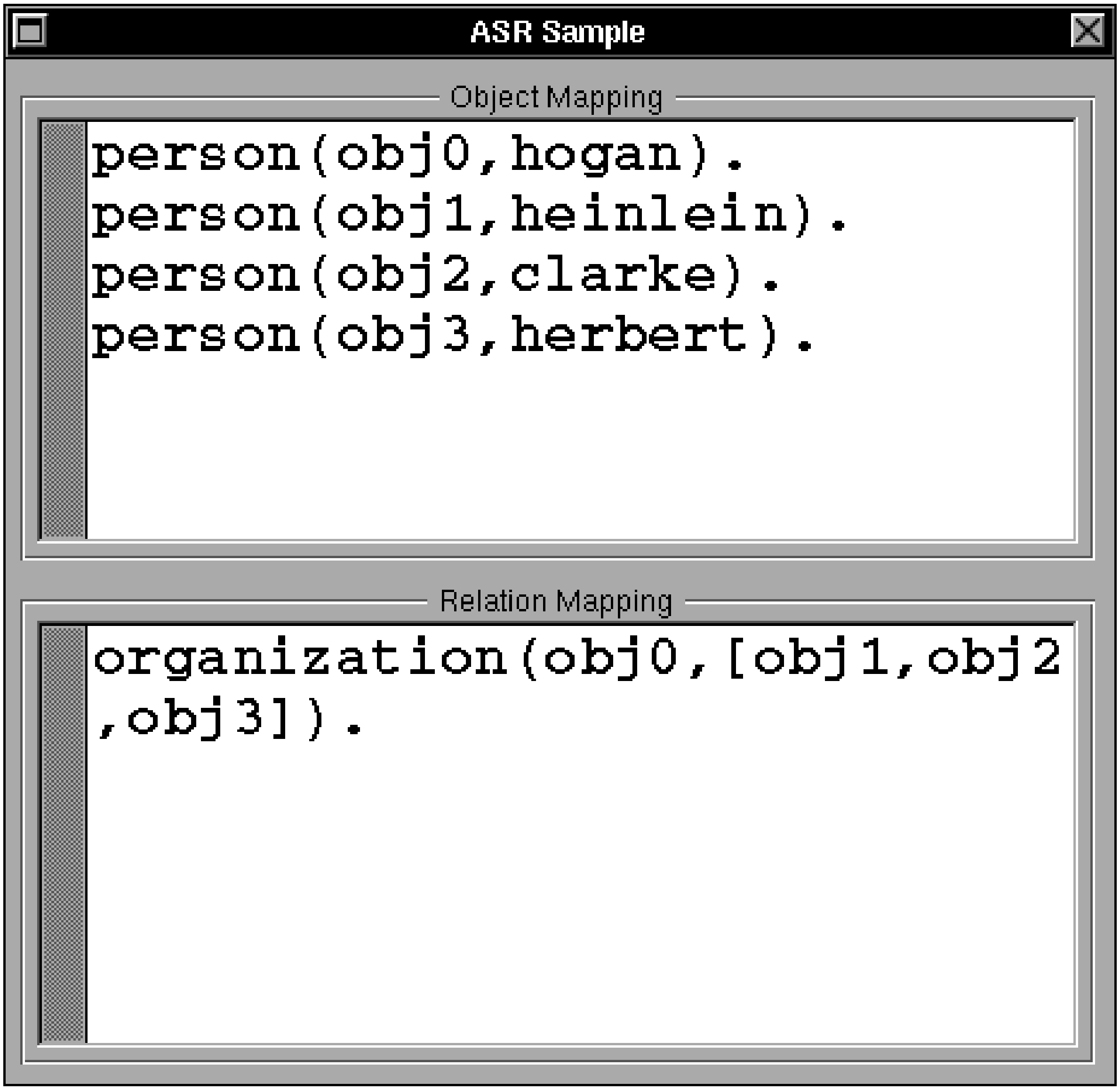


Figure 19: The ASR Sample Data

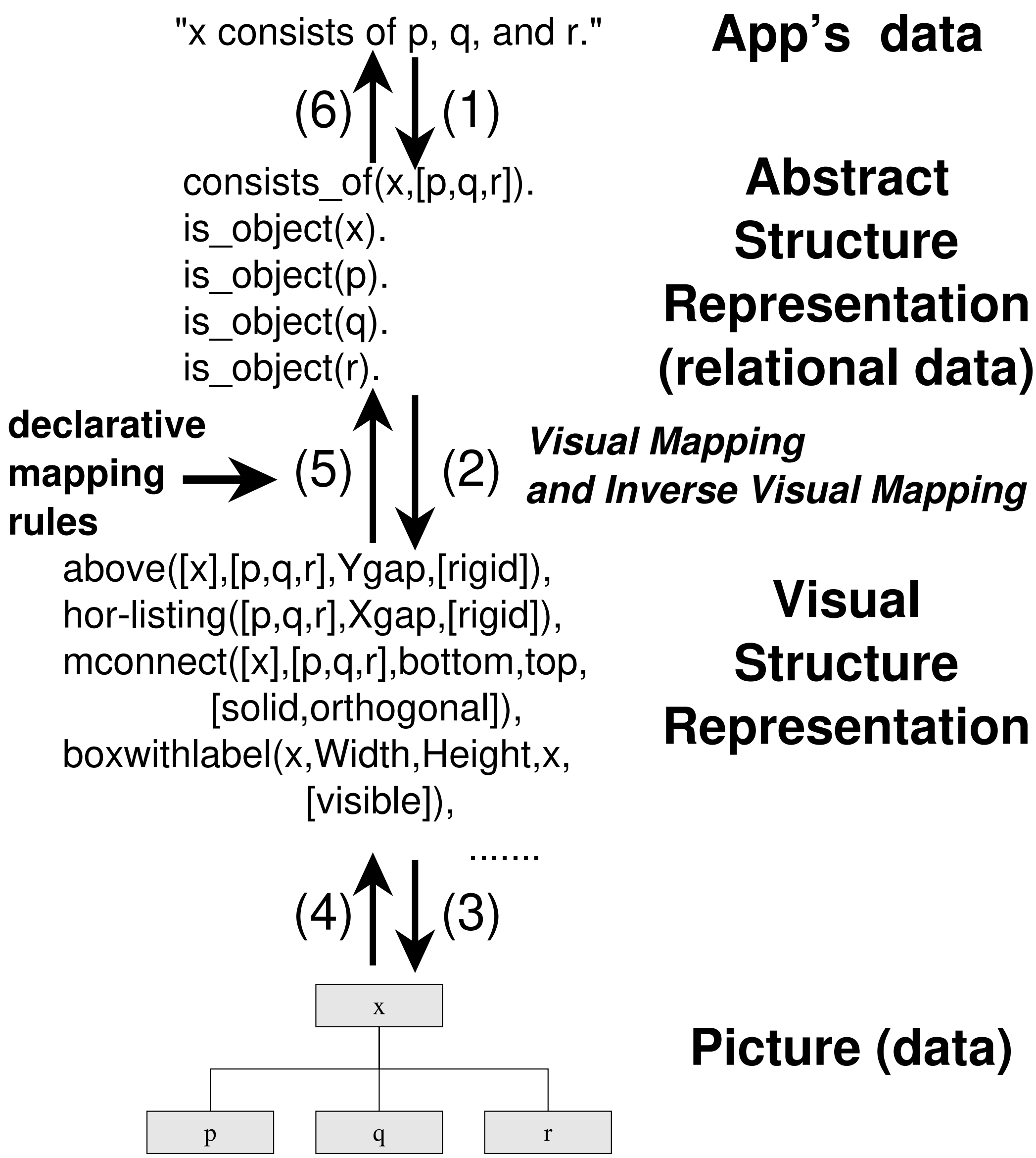


Figure 1: An Example of the Model

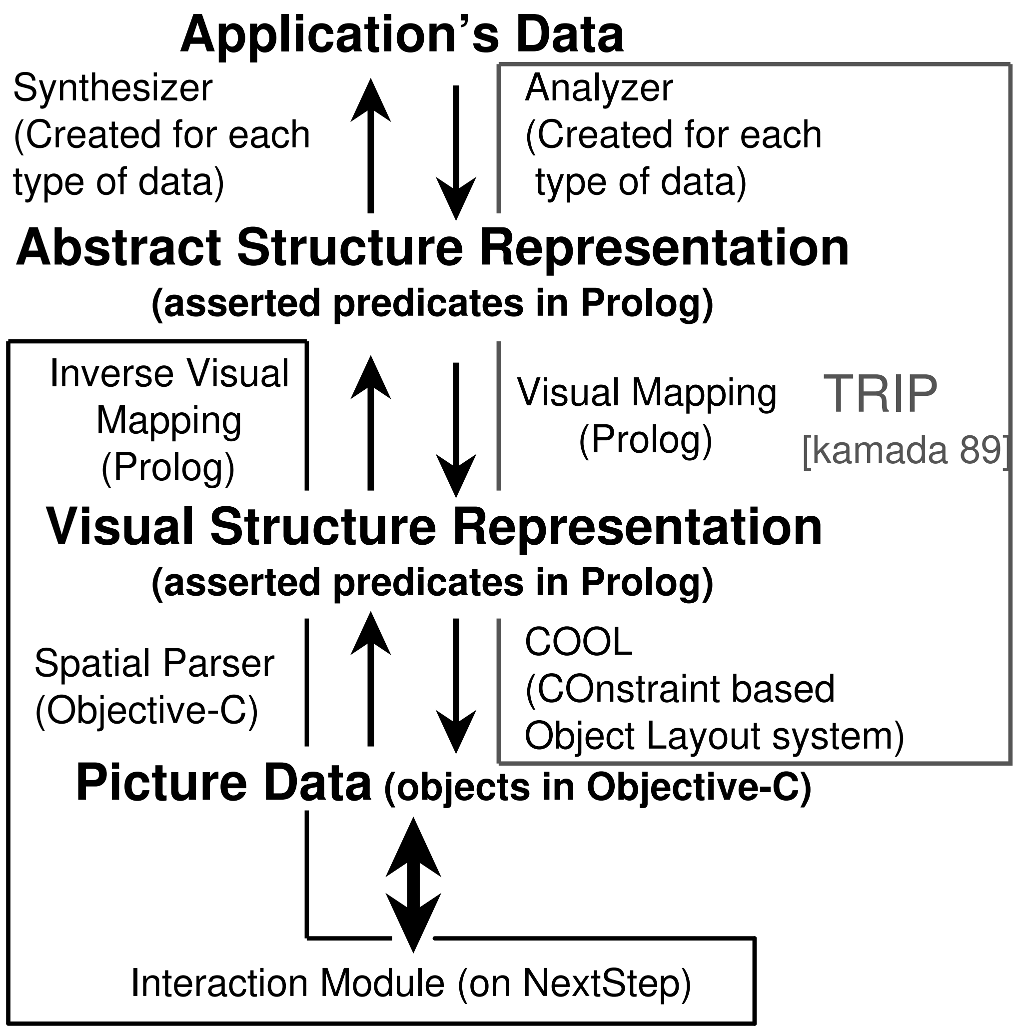


Figure 2: The Architecture of the TRIP2 System

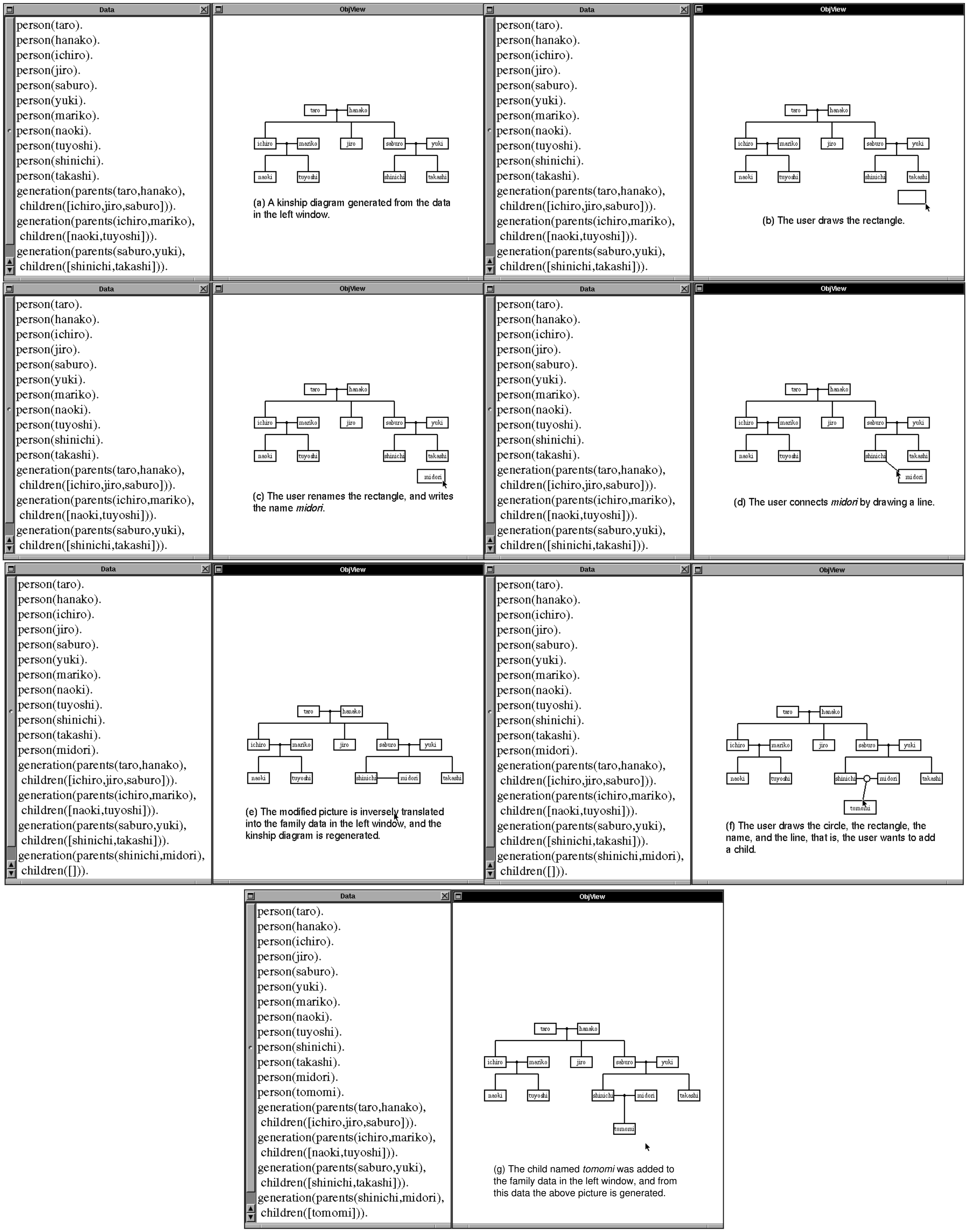


Figure 3: A Kinship Diagram

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

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

Figure 4: Visual Mapping Rules Example

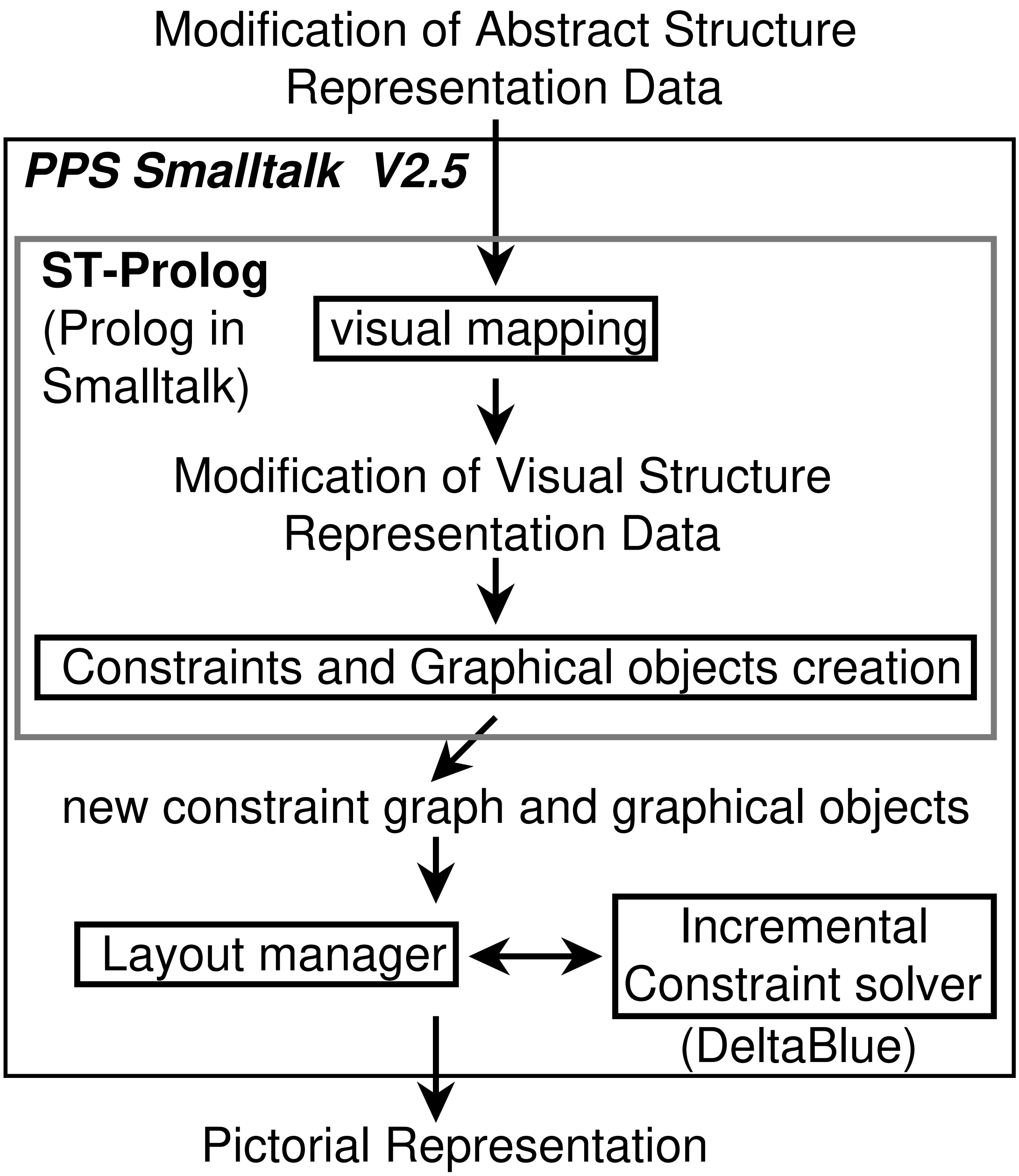


Figure 20: The DeltaTRIP system

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

Figure 5: Generated VSR Data

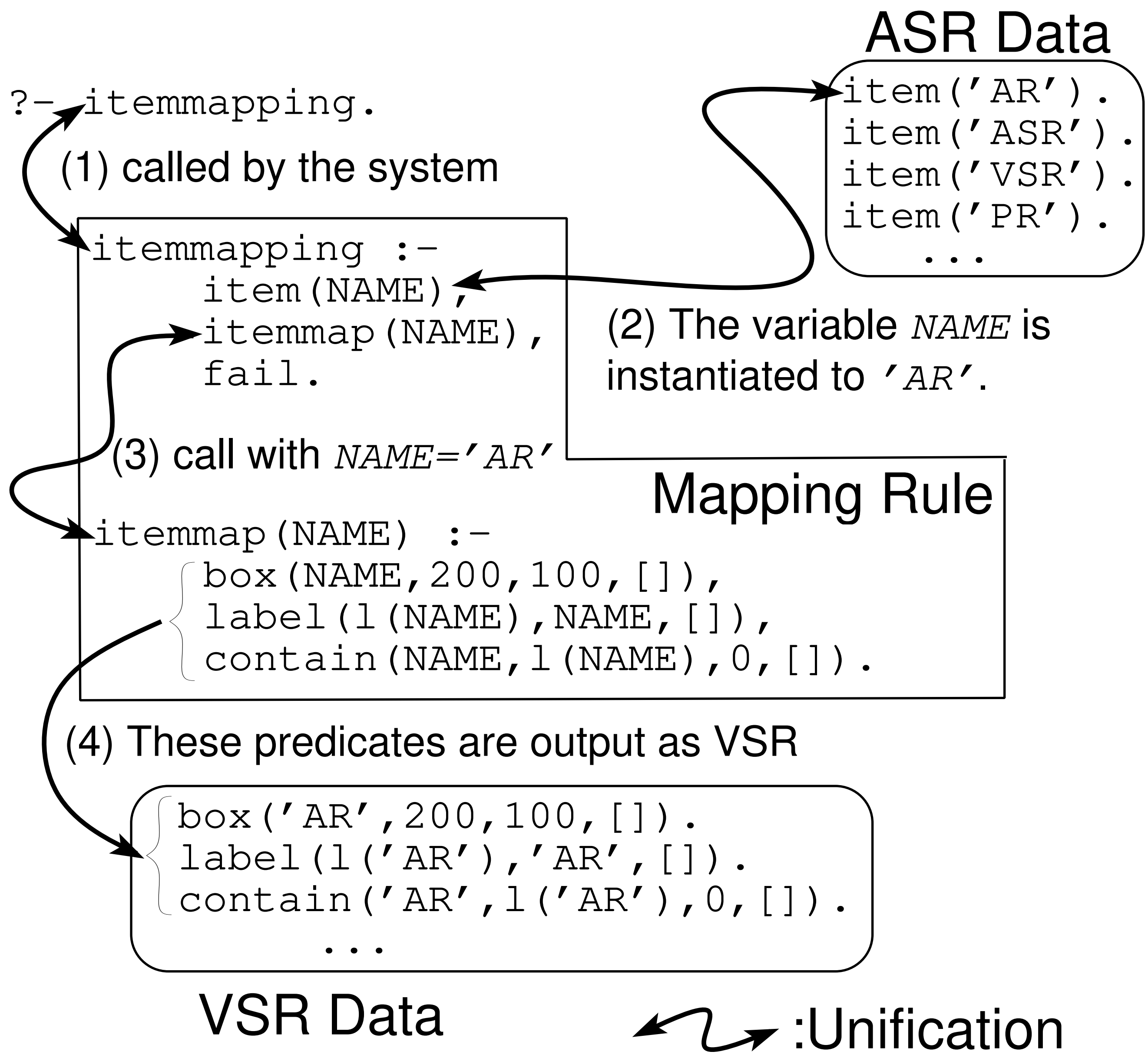


Figure 6: The Execution of Mapping Rule

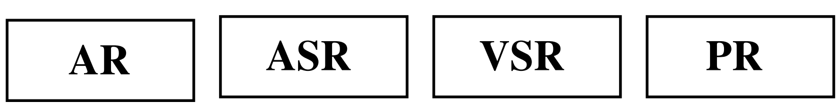


Figure 7: The Generated Picture

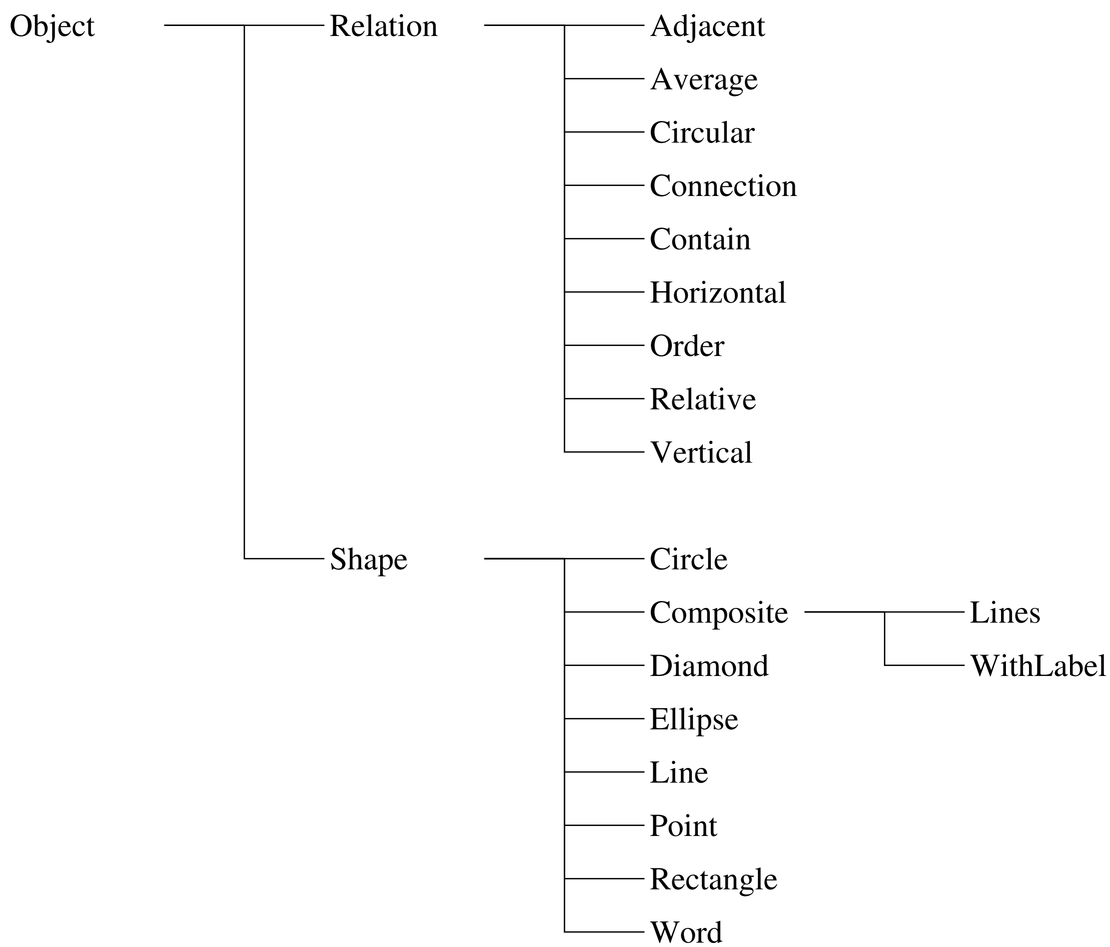


Figure 8: Classes in TRIP2

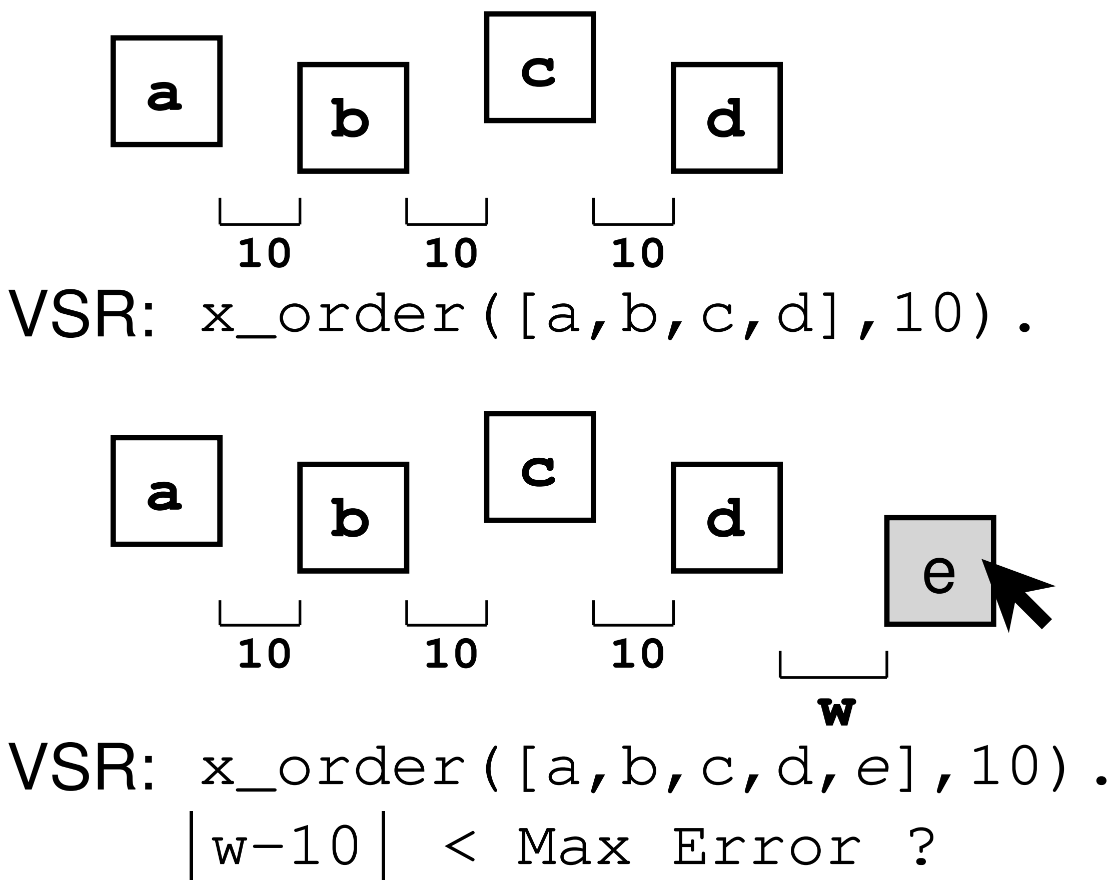


Figure 9: Adding a new graphical object to a relation

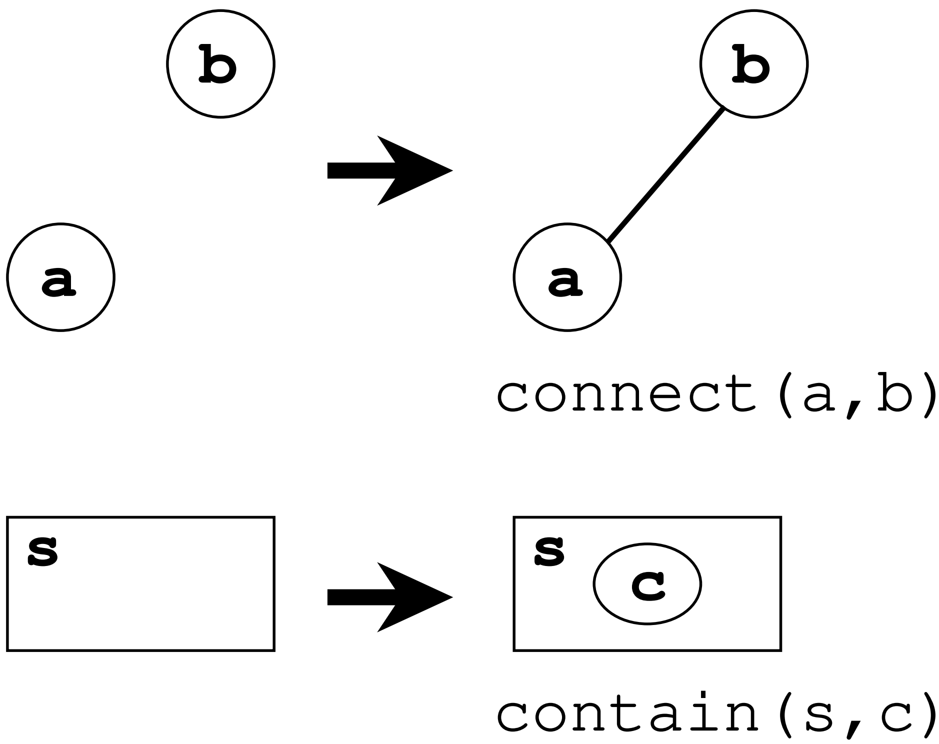


Figure 10: Creating a new graphical relation

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

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

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

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

Figure 11: Inverse Visual Mapping Rules Example

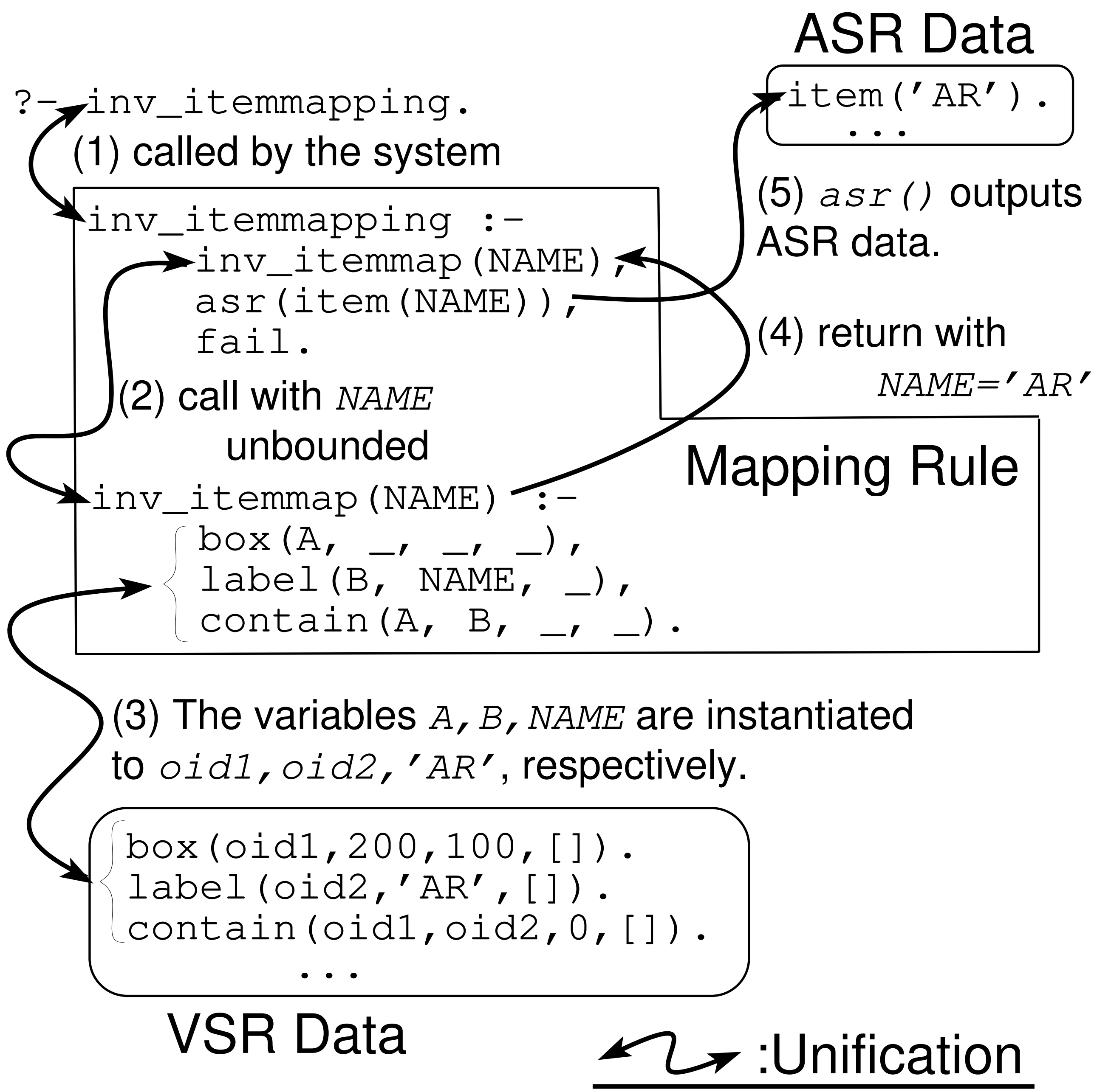


Figure 12: The Execution of Inverse Mapping Rule

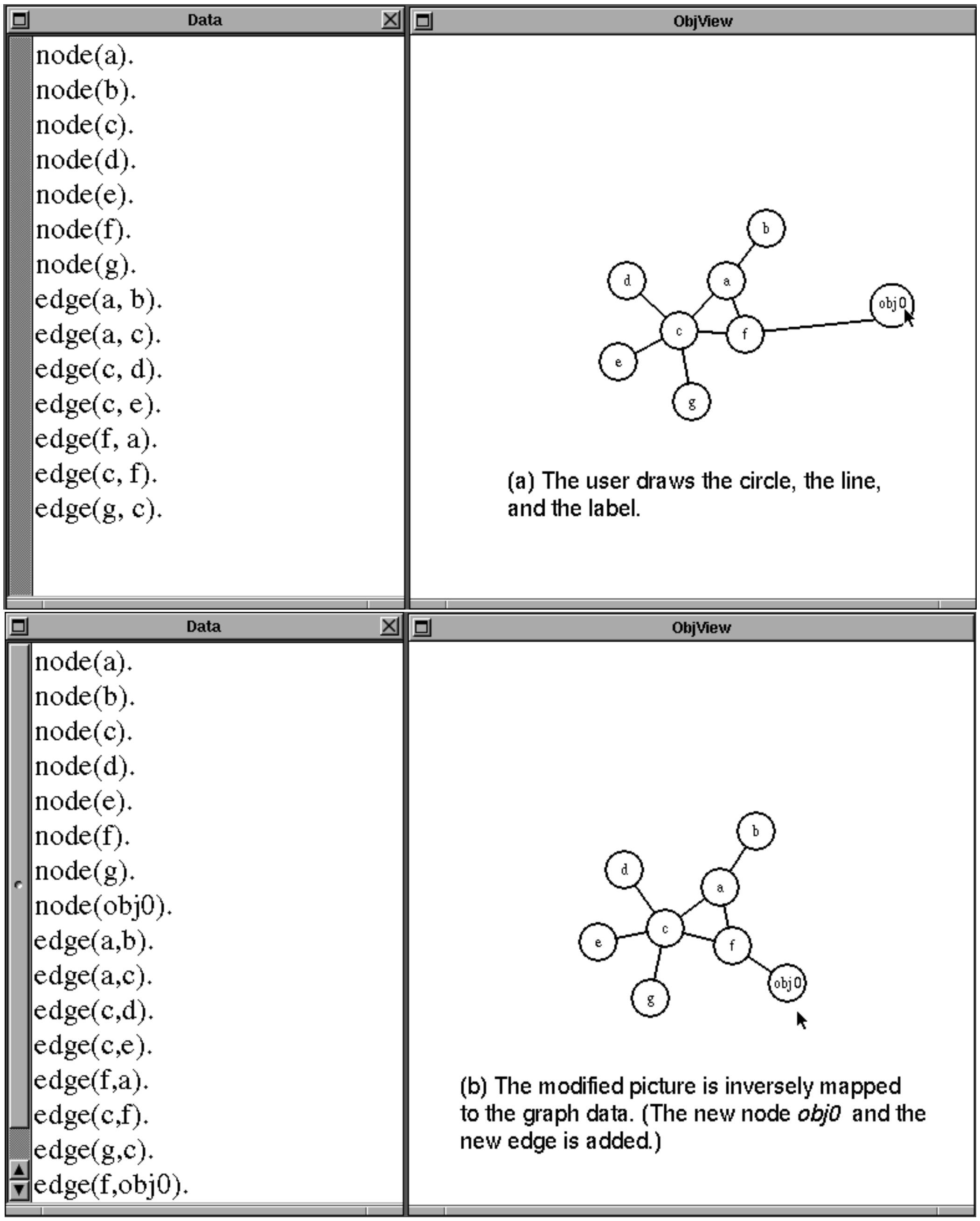
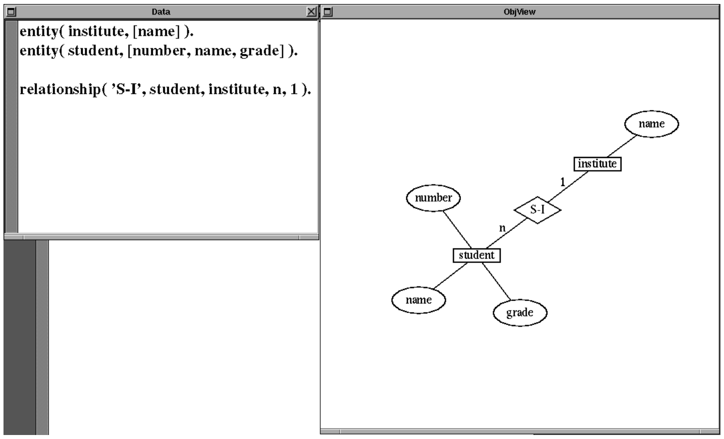
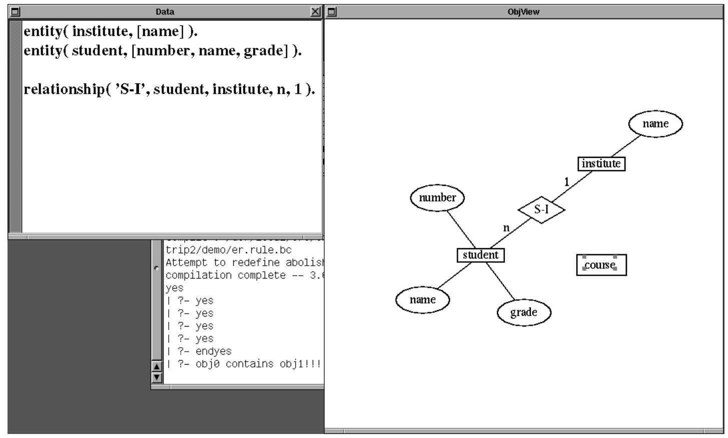


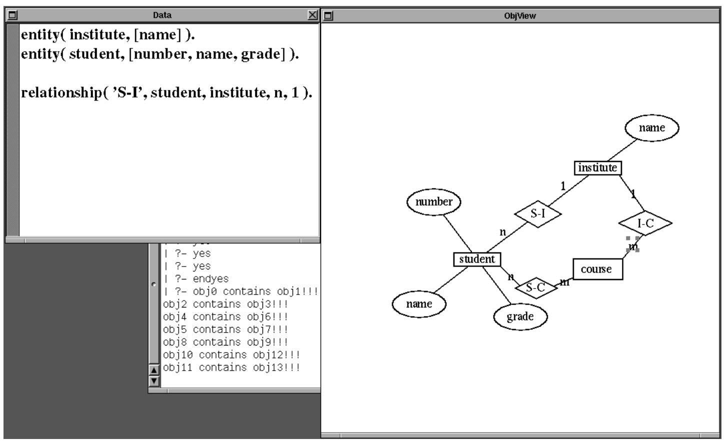
Figure 13: A Simple Graph Editor



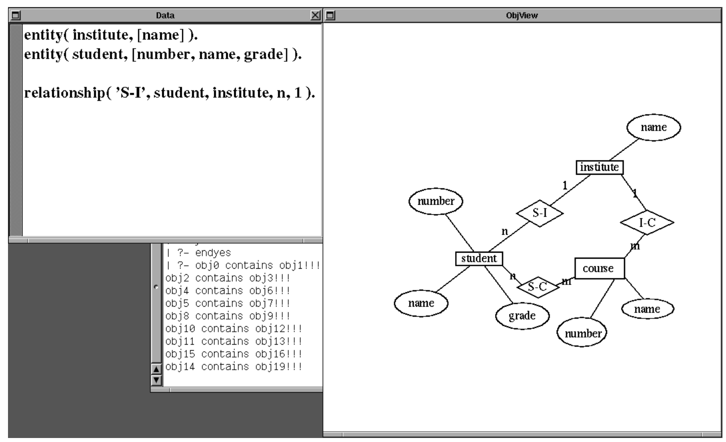
(a) The E-R database schema data in the left window are translated into the picture in the right window.



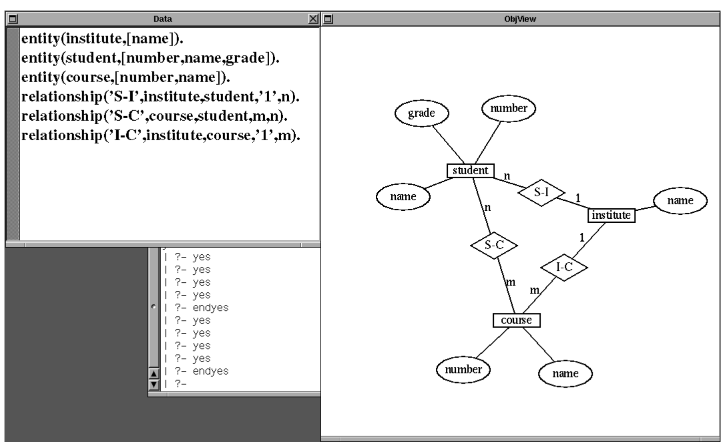
(b) The user draws an *entity* (a rectangle with a label) .



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

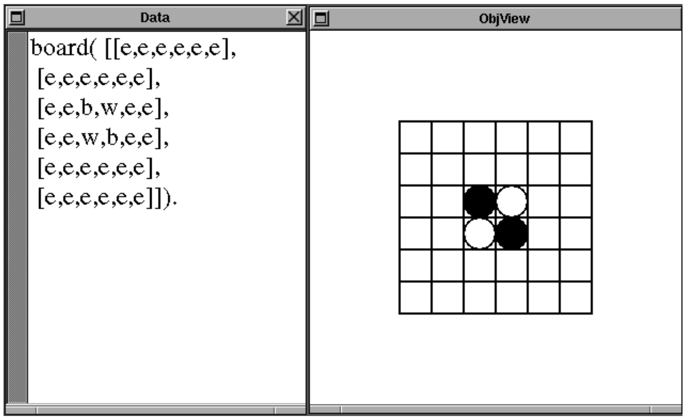


(d) Two *attributes* (ellipses) are added.

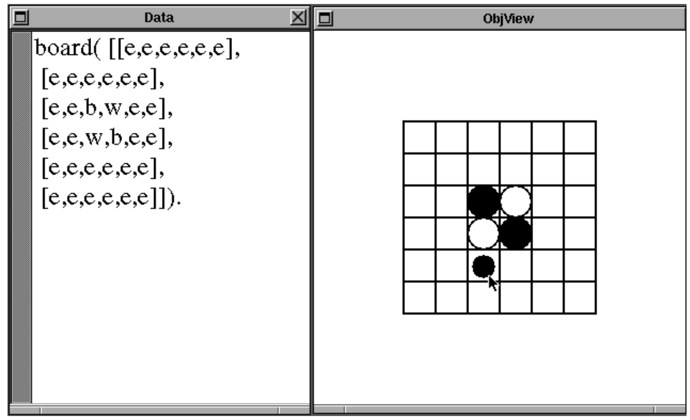


(e) The user executes the *flush* command. The updated schema is generated, and the E-R diagram is beautified.

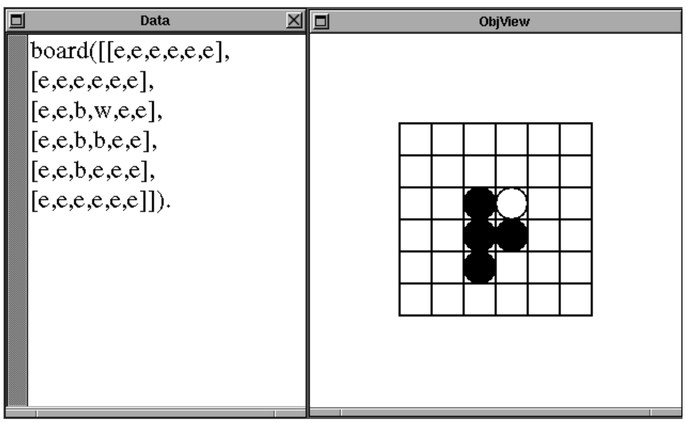
Figure 14: A Simple E-R Diagram Editor



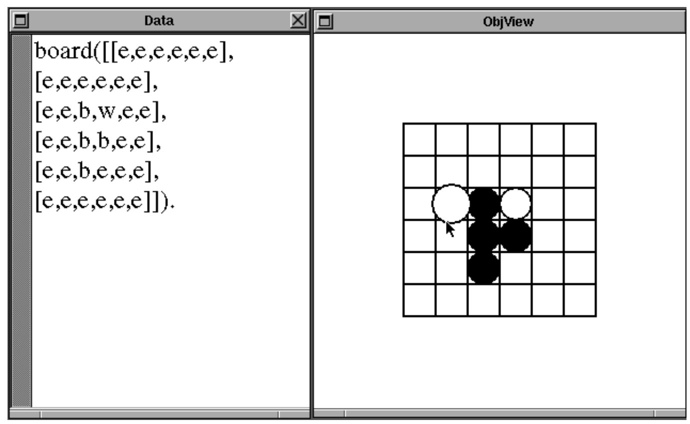
(a) The board-data in the left window are mapped to the picture in the right window.



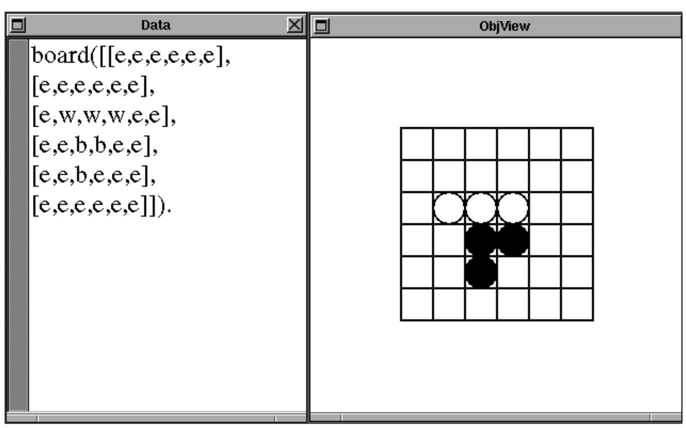
(b) The user draws a black circle. (note that certain margins are allowed for the size and the position of the piece the user draws.)



(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 15: TRIP2 Othello

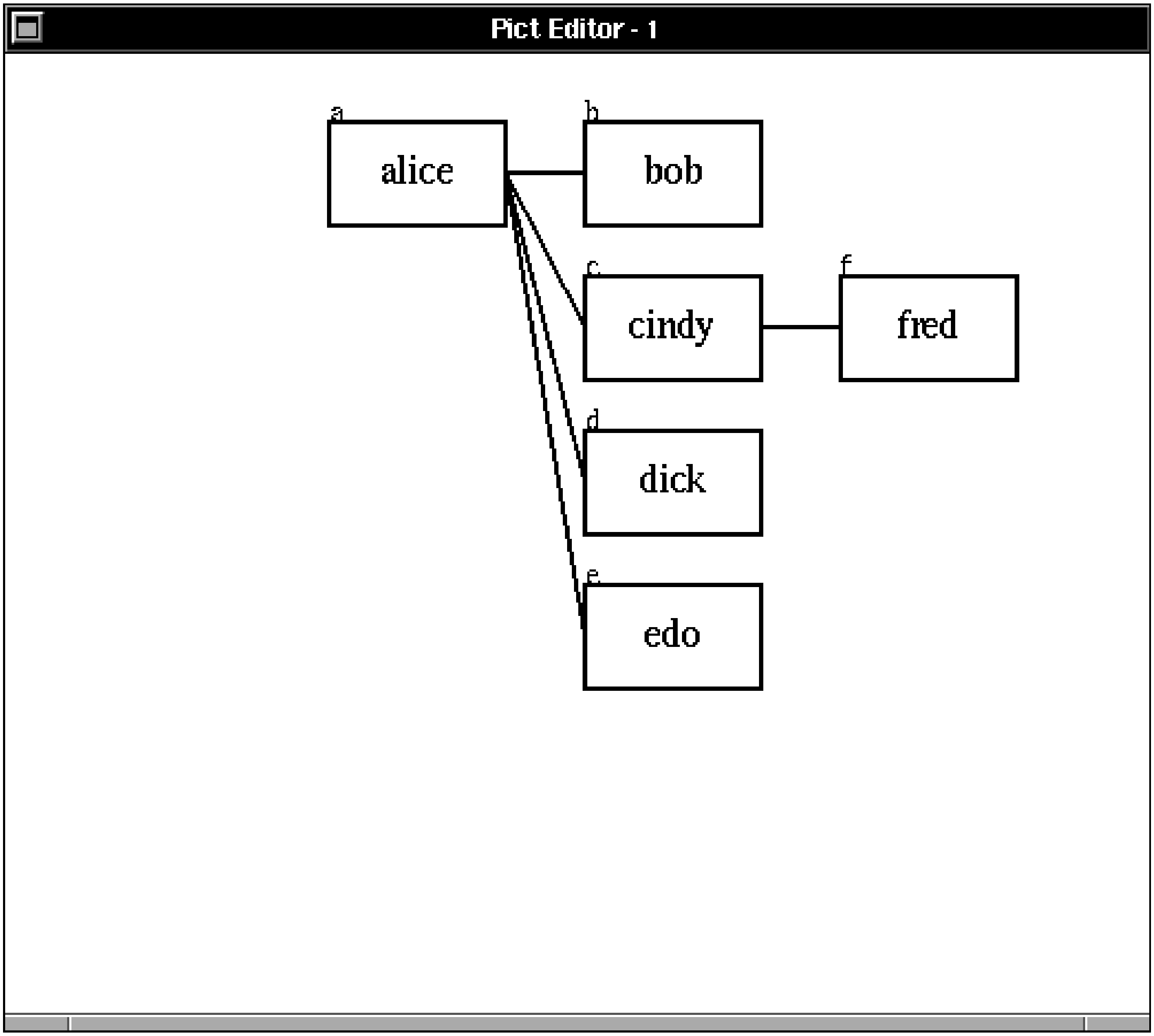


Figure 16: Organization Diagram

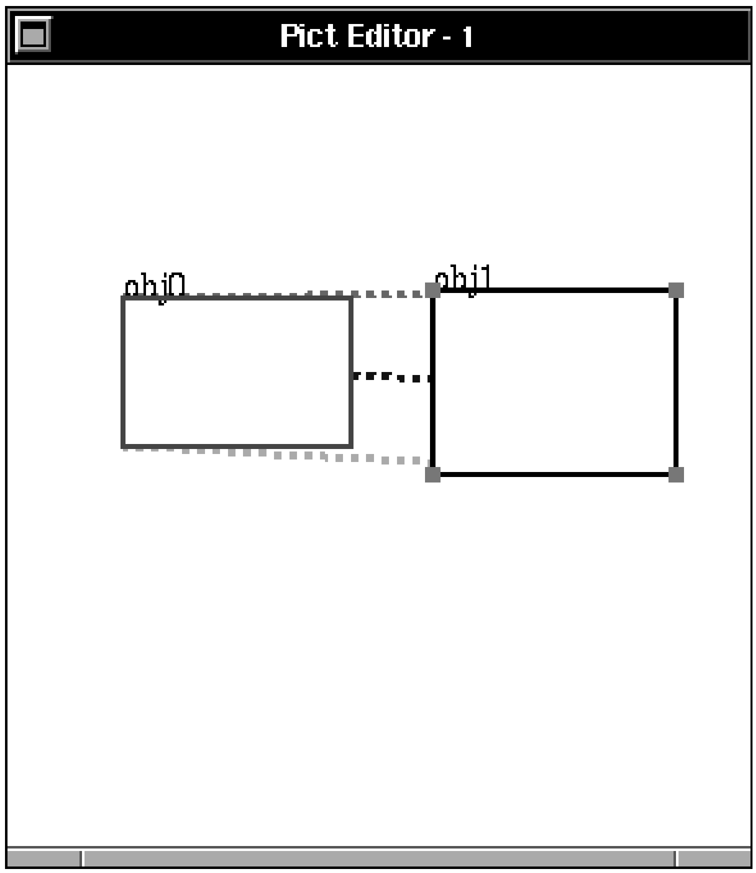


Figure 17: The Pict Editor

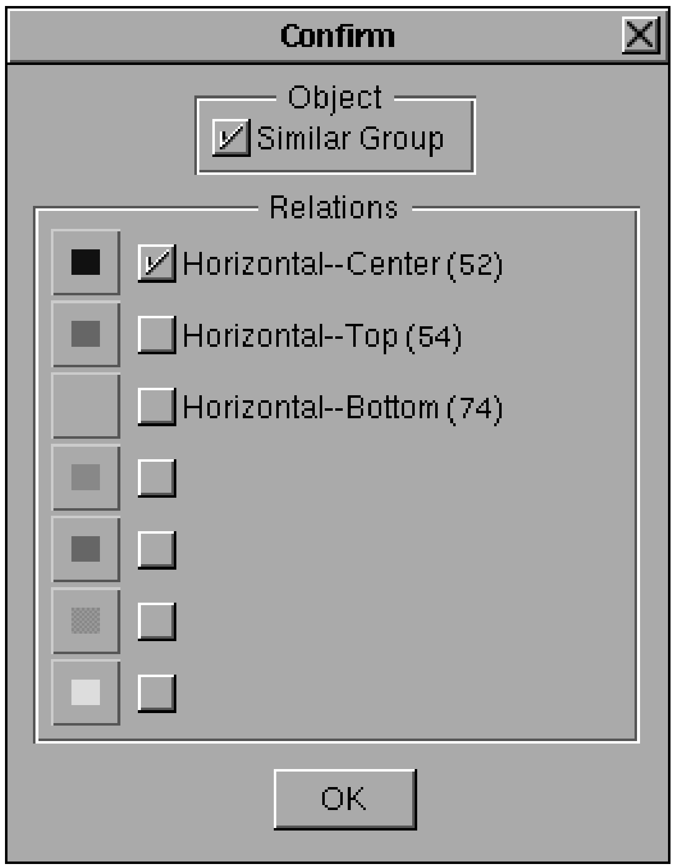


Figure 18: The Confirm Panel