Design of a Constraint-Based Hypertext System to Augment Human Reasoning

Bernard Bernstein, Paul Smolensky & Brigham Bell

CU-CS-423-89 January 1898

Design of a Constraint-Based Hypertext System to Augment Human Reasoning

Bernard Bernstein, Paul Smolensky &
Brigham Bell

CU-CS-423-89 January, 1989

Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
bernard@boulder.colorado.edu

Submitted to *Rocky Mountain Conference on AI, 1989*

Table of Contents

List of Figures

Design of a Constraint-Based Hypertext System to Augment Human Reasoning

Bernard Bernstein, Paul Smolensky & Brigham Bell
Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309-0430
bernard@boulder.colorado.edu

1. Introduction

This paper describes the design of a constraint-based hypertext system. The specific project is called EUCLID: a hypertext system to support reasoned argumentation. It is the result of exploration into human reasoning and analysis. We did not make any attempt at automatic reasoning since it is not an automatic system that we are after. We created an environment which augments human reasoning ability by providing a canvas on which the user may construct arguments and a palette of tools from which to choose.

We chose the domain of argumentation because we believe that the theories explored would apply to discourse in all fields, as Smolensky, Fox, King and Lewis argue in [5]. Research papers, decision making (e.g. reasoned letters to the editor), and even reasoned argumentation in conversation can all supply principles which are incorporated into the system.

In EUCLID, the canvas is the computer screen as controlled by a constraint-based layout system and the palette is defined in a *language* specific to argumentation called ARL (Argument Representation Language). This combination provides an environment in which users may design or analyze arguments at any level of abstraction that they feel comfortable with. Section 2 discusses how EUCLID was designed with this criterion in mind.

An additional objective of the project and this paper is to develop a powerful general framework in which hypertext systems may be defined. We chose a constraint-based approach to screen management and a database manager which allows domain-specific knowledge to be defined. The components of the generalized constraint-based hypertext system is described in section 4.

2. EUCLID

The EUCLID system is designed using a generalized framework for hypertext systems. This generalized structure allows the construction of domain-specific systems with very different interfaces, as illustrated in later sections of this paper. In order to recognize the significance of the system in terms of its usefulness as a general tool, we need to explore our motivation for developing it in the first place.

Spoken language, writing, and mathematical notation and proof are symbolic systems that have profoundly affected human reasoning capacity. Modern computers are powerful, active symbolic systems with the potential, we believe, to provide significant further advances in human reasoning ability.

A tool can be developed for helping people create and assess reasoned arguments and communicate these arguments to others. The tool must provide reasoners with a language, ARL, for expressing their arguments in a clear, precise, and relatively standardized fashion. The medium in which this language is realized is computer environment, EUCLID.

In EUCLID, the computer plays a role analogous to acoustic or print media in verbal or written argumentation: it provides a medium - an extremely powerful one - for supporting logical discourse among human users. We are *not* proposing to use computer reasoning to replace human reasoning. Our goal is to give users the expressive and analytic power necessary to elevate the effectiveness of their own reasoned argumentation.

2.1. ARL: Argument Representation Language

Our fundamental hypothesis is that in constructing an argument, two kinds of knowledge are brought to bear: knowledge of the subject domain, and knowledge of argumentation per se. These respectively manifest themselves as argument content and argument structure. A *general purpose* argumentation tool helps the user by virtue of its knowledge of argument structure, not argument content.

Drawing a clear line between structure and content is so crucial to this research that we find it useful to give that line a concise name: *the Divide*. Content information is *below* the Divide; structure information is *above* the Divide. Examples of assertions below the Divide are:

- Downhill skiing is thrilling.
- A working prototype is a necessary component of any research project.
- Approach Y to knowledge representation is seriously flawed.

Above the Divide we find statements such as:

- Claim C_1 supports claim C_2.
- Claim C is the main point of argument A.
- Claim C is made by author S.
- Claim C_1, made by author S_1 contradicts claim C_2 made by author S_2.
- Term T is used by author S_1 to mean D_1 but by author S_2 to mean D_2.

This latter sort of information is often not explicitly stated in text, but in it lies the structure that characterizes reasoned discourse. (The crucial importance of information above the Divide has also been emphasized by Zukerman and Pearl [6]. In the tutoring context, they have studied how such information is introduced through natural language expressions they call *meta-technical utterances*.)

Information below the Divide involves terms and predicates that vary completely from one domain of argumentation to another. But information above the Divide involves a reasonably constant vocabulary: the examples above use the terms *claim C*, *argument A*, *author S* and the predicates *supports*, *main-point*, *asserts* and *contradicts*. This vocabulary is characteristic of reasoned discourse in any domain.

ARL offers a set of primitive term-types and primitive predicates for formally describing argument structure, such as those mentioned in the previous paragraph. In addition, it incorporates high-order structures formally defined by combining simpler ones: for example, high-level standard schematic structures for arguments, arguments by analogy, allegations of misrepresentations, and argument refutations.

To give users the expressive power needed in real argumentation, ARL must let users extend the language's set of primitives and must provide the machinery for them to formally create their own high-order constructs.

The ARL statement corresponding to "Claim C_1 supports claim C_2" uses the formal predicate **supports** to relate two entities that have formal type **claim**. The content of each claim is not expressed formally, but *informally*, in natural language. For example, the content of C_1 might be "Lower interest rates lead to bull markets." Thus ARL is a *semi-formal* language: argument *structure* information (above the Divide) is represented *formally*, while argument *content* (below the Divide) is represented *informally*. The computer has access to the semantics of the formal information, but only the user has access to the semantics of the informal information.

We believe that the notion of semi-formal language is potentially of great value to the design of effective joint human/computer systems: its applicability extends beyond joint human/computer reasoning, to include nearly any joint human/computer activity.

2.2. The EUCLID environment

A formal representation of the structure of the argument contained in a theoretical research paper is too large for anyone to explicitly represent without the help of a data manager. The computer environment EUCLID keeps track of ARL representations, allowing users to select portions of the argument to be displayed on a high-resolution graphics terminal. In addition to displaying ARL structures, EUCLID allows users to add new structures, and modify old structures. Users can state that they want to create a new instance of a higher-level construct (say an analogy), whereupon EUCLID prompts the user for the necessary inputs and manages the details of creating the necessary data structures. Information is displayed in ways specially designed to show the various argument components. For example, there are special displays for analogies, refutations, or retrieval requests like "show all claims whose validity depends on this one." Along with the capability to create new types of argument structures, users have the capability to specify new ways of displaying them. Users also have considerable choice among alternative types of displays.

3. Constraint-Based Hypertext

Hypertext has drawn attention from many fields of study because of their usefulness in managing large and complex databases [4]. A generalized framework for managing hypertext systems would be a valuable asset to the community who are developing these tools. Our approach to providing an environment for hypertext construction is by dividing the system into many independent units. Each unit is independent from the rest of the system to an extent, so as new applications require different functionality, they may be integrated without great reconstruction effort.

In conventional hypertext systems, there is no or little support for the display of the information. It is often the user who is given with the task of finding a useful layout of the data. There may be cases when only the users know exactly how their data should look, and they would not want any outside assistance in organizing it. On the other hand, many visual representations are useful; in fact, more information may be provided by virtue of the layout of the information [3].

The reason we call the system "constraint-based" is due to its method of performing screen layout. Relationships between data are encoded in a database system, but the visual representation of the relationships must be made visible to the user. Our method of making the transformation from database to screen was by making an analogy between relationships and constraints. Constraints in our system consist of spatial relationships between graphical objects, just as relationships in the database represent domain-specific connections between data.

Using this analogy, we are able to make the transition from the database to the graphical representation of the data and the relationships between data. With a constraint-based approach to layout management, the mapping between database relations and constraints need to be clearly defined. With a two dimensional output device (such as a computer terminal), we are limited to the number of items which may be adjacent to each other, therefore, there need to be constraints which allow other methods of representing relations.

3.1. Examples

Before we discuss the specific components of the system, we will see some examples to illustrate the flexibility which constraint-based hypertext systems can provide.

3.1.1. Outliner

A simple application for a constraint-based hypertext system would be to create a standard outlining program. Levels of information may be denoted by their indentation from the previous level. The only relation which needs to exist is **child**, which represents a *sub-category*, *component*, or any other similar hierarchical relation.

A constraint would then be associated with the relation which visually depicts a representation of child. This constraint may be defined differently depending on the users preference. In the examples shown in figures 3-1, 3-2 and 3-3, we demonstrate three possible representations of the same outline. The elements are the same and the database is the same, but the constraints have changed from one layout to the next.

3.1.2. The definition of EUCLID

Using our generalized framework for constraint-based hypertext, we are now able to produce a definition of EUCLID in terms of our system.

The relationships which we will represent in this example are *supports*, *refutes* and *elaboration*. Supports will be represented by a constraint which puts a supporter below and indented from the original

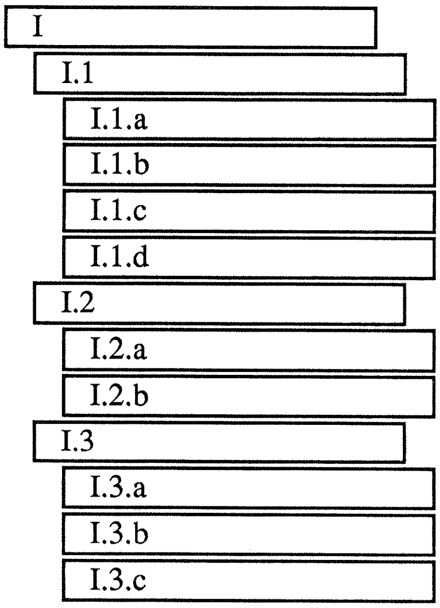

Figure 3-1: Outliner: Children are constrained to be below and indented from the left of their parent.

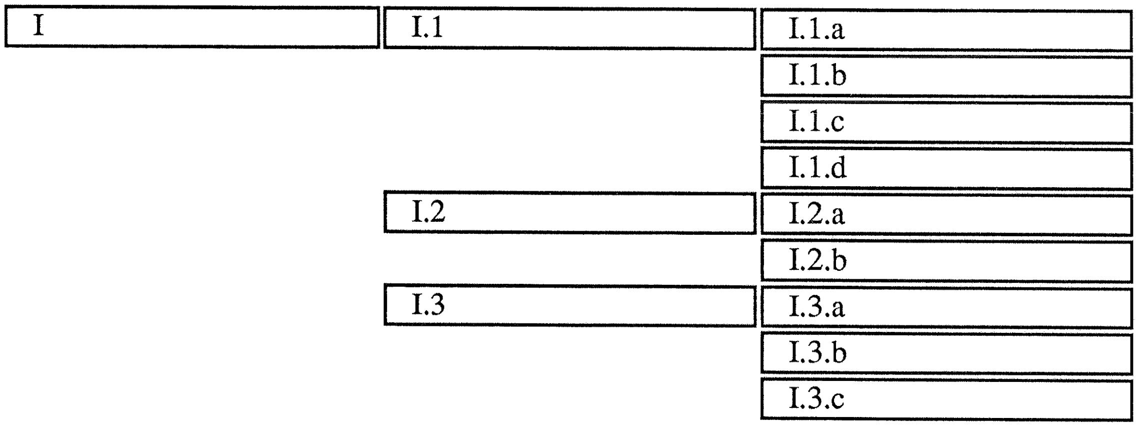

Figure 3-2: Outliner: Children are stacked to the right of their parent.

statement. Refutations will be to the right and elaborations to the left.

Directed edges are added to the display so that some more information about the relationships may be apparent from the display. Figure 3-4 illustrates a layout given by this system.

4. Components

In section 3 we discussed some of the attributes and components of constraint-based hypertext, but did not feel that the details could be explored until the basic ideas are understood. Now that some examples have been given, we can continue with a more detailed description of the system components.

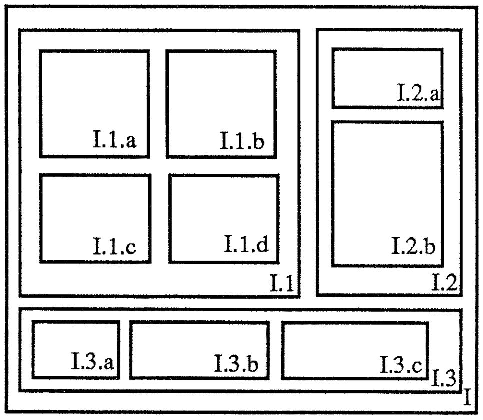

Figure 3-3: Outliner: Children are placed inside parent.

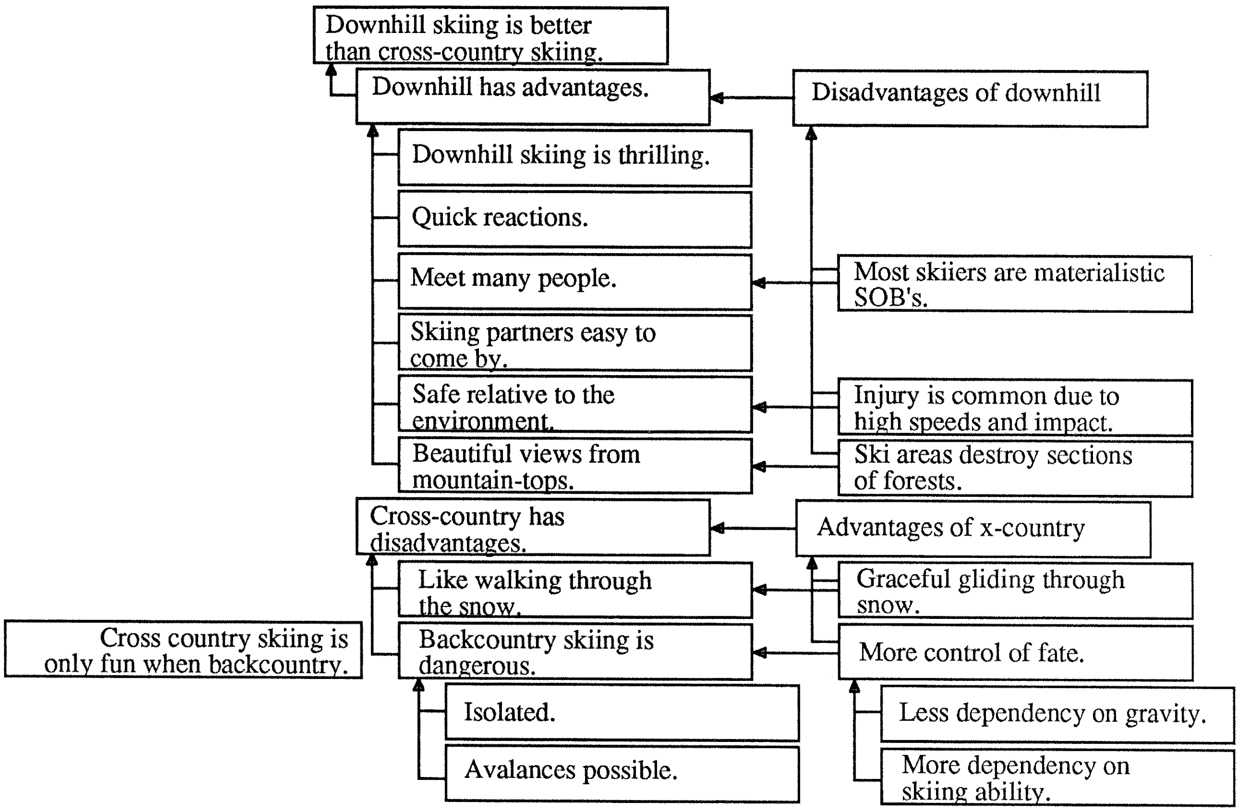
Downhill skiing is better than cross-country skiing.
Downhill has advantages.
Disadvantages of downhill
Downhill skiing is thrilling.
Quick reactions.
Meet many people.
Most skiers are materialistic SOB's.
Skiing partners easy to come by.
Safe relative to the environment.
Injury is common due to high speeds and impact.
Beautiful views from mountain-tops.
Ski areas destroy sections of forests.
Cross-country has disadvantages.
Advantages of x-country
Like walking through the snow.
Graceful gliding through snow.
Cross country skiing is only fun when backcountry.
Backcountry skiing is dangerous.
More control of fate.
Isolated.
Less dependency on gravity.
Avalances possible.
More dependency on skiing ability.

Figure 3-4: An example of EUCLID.

A constraint-based hypertext system consists of several important components which are necessary to implement such a general purpose environment. The components may be described horizontally (everything necessary for the front-end) or vertically (all components of each section).

Horizontally, we can represent the system as in figure 4-1, and vertically we represent the system as in figure 4-2. First a description of the horizontal architecture will be given, then the vertical.

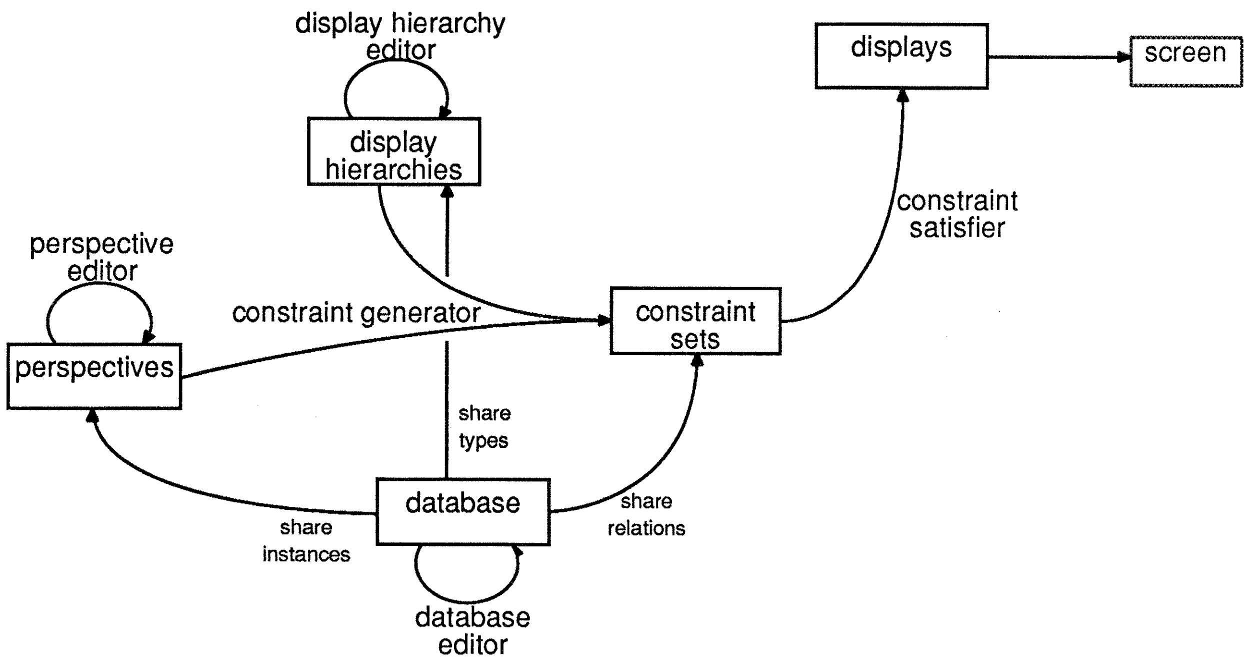

Figure 4-1: Horizontal representation of the constraint-based hypertext architecture.

In figure 4-1, we will begin at *database* and work upward to *screen*.

The *database* contains the domain information and instantiations of databases within that domain. ARL is one such database, and the instantiations are furthermore present in this section. The instances in the database are shared with the perspectives, the types in the database are shared with the display hierarchies and the relations are shared with the constraint sets.

A *perspective* is a subset of the database that is displayed on the screen all at one time. Elements in a perspective are selected by users based on their interpretation of the database, or their own perspectives of the subject. Different perspectives of the database may coexist and are instantiated depending on how the user wishes to view a portion of the database. At any one time, one perspective is **active**: this is the one currently being displayed.

Display hierarchies are hierarchies of constraints which may be used to determine layout of individual elements in the active perspective. The hierarchy is based on inheritance of types in the database, and determines how elements of differently types will be displayed. If the element consists of a set of relations, then a set of constraints needs to have been either inherited from a parent type or associated with the given type.

Constraint sets are the constraints which are used to compose the screen layout after all the units are instantiated and the relations are defined. The relation types determine which constraints are used to motivate the overall layout of the screen. The relations constraints along with the constraints issued in the display hierarchies are combined to give the complete set of constraints for the current perspective. The *constraint satisfier* is then used to generate a *display*. This is then observed on the *screen*.

The architecture in this form is useful for the general description of the system, however in order to consider the actual components of the environment, the vertical representation would be more suitable as in figure 4-2. The following sections will describe each component of the system in some detail.

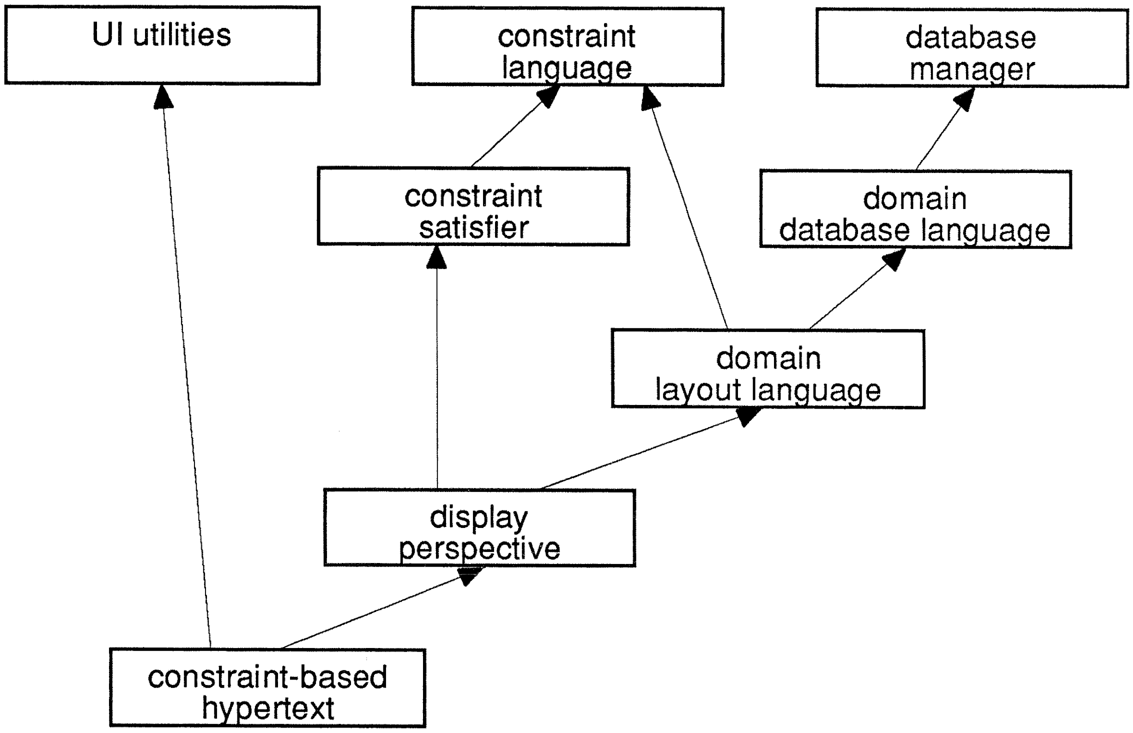

Figure 4-2: Vertical representation of the constraint-based hyptertext architecture.

4.1. Constraint Language

A constraint language is used to encode constraints. The language is general enough to permit coding of constraints which may be satisfied by an independent solving system.

A constraint language must be able to define constraints on spatial relationships between objects. *Hard* equality constraints should be supported to allow exact placement between objects, and another form of constraint must exist to permit resolution of conflicts between constraints.

A problem with defining generalized hard constraints is that of dealing with conflicts. If conflicts may not exist, then constraints may simply be a matter of setting values to variables. For example, if a constraint exists, such as:

```
    to-right(a,b)
```

which is defined as

```
    left-side(b) = right-side(a)
```

then as long as no other box is to the right of *a*, the value can simply be set accordingly. This simple case is made more complicated, however, when there are other constraints in the system and when there are conflicts between objects with the same constraints.

Because of these problems, we needed to incorporate some method of allowing for conflicts. One method is by implementing *soft* constraints, another is to allow hard *inequality* constraints. In our current system, we are using inequality constraints: constraints which suggest that variables be greater than or less than other variables to allow movement in one dimension at a time to avoid overlaps.

To repair the above constraint we can use a variation on the same constraint encoded using an inequality:

```
    left-side(b) = {x | x > right-side(a)}
```

In this case, if there is a conflict at evaluation-time, then it may be resolved by changing *x* until it is to the right of all solved dependents of *a*. Another, method of resolution would be to increase *x* until *b* is not overlapping anything. This forces some more work at evaluation time, but it will guarantee that conflicts will not occur.

Another method of resolving conflicts would be by using an error function to determine the *goodness* of the constraint instance. This determines how close to an actual solution the constraint is at any given time. Using this method would give us a soft constraint system. It would allow violations in constraints, yet still maintain a quantitative assessment of how violated the constraints have become. Numerical optimization techniques could then be used to minimize the overall error function for the system. This approach would produce a very generalized system, however as the number of constraints increase, the complexity of the optimization problem grows quickly, and it would not take many objects to slow the system down significantly.

Other components for a good constraint-based system are described in [2, 1]. We used a subset of the ideas to reduce the overhead and increase the efficiency needed for a usable system.

It was decided that some constraints need to be equality constraints, setting specific values, and others need to be soft or inequality constraints. The equality constraints represent the relationships between objects, and the inequality or soft constraints allow space for conflict resolution. Since the screen is two dimensional, it seemed reasonable that at least one of the dimensions should be assigned an equality constraint so that some relationship can be seen between objects.

As an example of the constraint language, we will discuss the definition of the constraint **left-justified-below**. First we see some constraint definitions of one dimensional constraints, then **left-justified-below** is defined which calls the two prior definitions. This example uses only hard constraints: inequality in **below** and equality in **left-together**.

```
define_constraint below (top_box, bottom_box)
  top(bottom_box) = {x | x > bottom(top_box)}

define_constraint left-together (boxa, boxb)
  left_side(boxb) = left_side(boxa)

define_constraint left-justified-below (top_box bottom_box)
  below(top_box, bottom_box)
  left-together(top_box, bottom_box)
```

An example of an error function for the constraint **below** is given below. This is not used in the same system that the above constraints were defined in. Here we define an error function as used in a soft constraint system.

```
if (top(bottom_box) < bottom(top_box))
  then
return infinity
  else
return ((top(bottom_box) - bottom(top_box)) ** 2)
```

In the case of the hard constraints, where an arithmetic function is performed, the form is simply evaluated to get the new value. If an inequality is used, then a value must be found which satisfies the inequality. The method for performing this resolution is dependent on the constraint solver. An example of an arithmetic function would be if **left-justified-below** were extended to include an indentation of the lower box as illustrated in the Outliner system (section 3.1.1). The constraint of **left-together** could be defined as:

```
define_constraint left-together (boxa, boxb, indent)
  left_side(boxb) = left_side(boxa) + indent
```

4.2. Constraint Solver

There are many methods of satisfying constraints. The system which provides this functionality may be somewhat independent from the constraint language. The language defines the constraints, and the solver works at run-time to find the solution to the constraint problem.

There are three problems which the solver must be able to handle. Satisfying constraints, resolving conflicts in constraints, and resolving less immediate conflicts such as overlaps.

Constraint satisfaction of equality hard constraints consists of setting the value of a dependent variable to the value given in an equation containing either independent variables or variables which have already been satisfied.

If soft constraints are used, then an algorithm will use the error functions of the individual constraints to find an overall error for the system. This will determine how *good* the layout is and changes are made (depending on the approach) in an attempt to reduce the overall error. Local minima often exist in these layout problems, so steps must be taken to avoid them. If inequality constraints are used, as described in 4.1, then a different approach is used.

Using numerical methods of optimization buys us a solution to conflict resolution between constraints if our constraints are defined well. If two objects are given the same constraint and furthermore they are constrained not to overlap each other, then a solution will produce a compromise in one constraint to provide for the resolution of the other constraint. The problem of resolving more distant conflicts is a matter of using more constraints to fix the problem. Using *not-overlapping* as a constraint defined between every pair of objects would resolve the problem in the general case.

With inequality constraints, a function is used to find values which satisfy the given inequality without producing conflicts in the overall system. At this level, it is convenient that our solver has some basic knowledge of the system. Knowledge we need is that of what a conflict consists of. In some cases, an object may wish to reside inside another object, while at other times, objects may avoid overlap altogether. If the condition that we are avoiding is overlapping objects, then we can increase (or decrease in the case of less-than constraints) the value in question until the constraint is satisfied and the object being moved is no longer overlapping other objects. This will avoid overlaps in general, however it does not provide an optimal solution.

The method used to resolving overlaps consists of first determining the cheapest (smallest change in a variable) way of resolving the overlap and then determining who can provide the variable change. In reference to figure 4-3, we see that the possibilities for resolving the conflict are by moving nine units in horizontally or five units vertically.

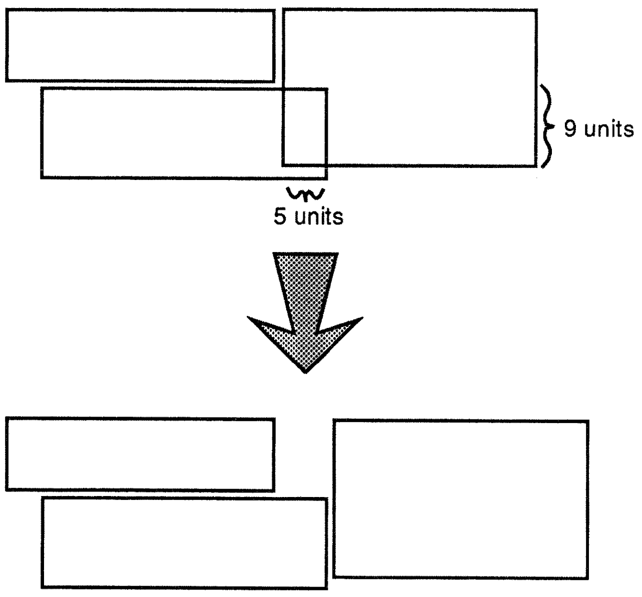

Figure 4-3: Overlap resolution

Choosing the lesser of the two, the algorithm decides that the large object to the right is able to move because it has an inequality constraint in that dimension. If the object was constrained with an equality constraint, however, it would not be able to move horizontally, so objects which the object in question depend on would be checked for horizontal movement. If movement is still not possible, then the other box would be moved vertically to resolve the conflict.

4.3. Database Manager

The database management system must be able to represent objects and relations as in a semantic database. The objects in the database correspond directly with the objects which are displayed, although not necessarily all objects are displayed. The relations contribute display constraints to perform layout, but like the objects, they are not all necessarily in use unless the related objects are being displayed. This relationship between the database and the display are discussed in 4.5.

A facility for implementing composites may be useful depending on the application of the system. In EUCLID, we use composites to define arguments which consist of sets of relations and which provide their own display constraints.

4.4. Domain Language

In order to make a generalized system usable, some domain knowledge must be encoded. In the EUCLID project, we developed ARL to provide knowledge of argumentation. This language is actually a database language which has names that are useful for the domain of reasoned argumentation.

Within the conceptual level of the domain language, there are two sub-levels which must be distinguished: the **domain database language** and the **domain layout language**.

The domain database language has three main components: **nodes**, **relations**, and **composites**. Nodes are the objects which contain uninterpreted data such as text or graphics. Relations are objects which give links between nodes. Composites are sets of nodes and relations which define larger structures such as arguments.

The database language allows the definition of data and relation types. The definitions are instantiated such that the database manager may perform queries and maintain persistence of the information. The database language does not provide any display layout information.

In the domain layout language, constraints are used to enforce layout of the relationships between nodes. At this level, a distinction must be made between fixed-arity and dynamic arity relations. Fixed-arity relations allow the nodes involved to be given constraints a priori, but if the arity is not known, then the constraints need to be more general to accommodate more objects.

A binary relation such as *supports* can use a constraint which forces one object to be below another object, but an unspecified arity relation such as *conjunction* may have any number of objects related to each other. Another object is created as a "placeholder" which represents the relation per se, so that the elements in the relation may be constrained to it.

4.5. Display Perspective

This is a small section of the system which manages the objects and relations which are being displayed at any moment: the active perspective. This is a subset of the entire database, and may include the entire database if the database is small enough. The ability to hide objects from view while retaining their integrity in the database is the main function of this manager; this is achieved by removing objects from the active perspective while retaining them in the database.

The display perspective may also provide persistence. Any set of objects that are currently being displayed could be saved as one particular perspective, so that in future sessions, various perspectives may be retrieved and displayed according to the section of the database that is desired.

4.6. User Interface Utilities

In order to have a complete hypertext system, we must have a user interface which provides the ability to use direct manipulation on the objects in the database. The use of a mouse as the pointing device allows the user to specify objects by simply pointing at them. As a direct manipulation interface, the user may do operations on the objects in the database as if they were objects in the "real world".

5. Conclusion

The EUCLID project provides users with a framework designed to enhance their own reasoning ability. It does not solve problems for its user, but rather maintains a consistent problem-solving environment which organizes arguments graphically and keeps the information persistent. Our immediate objective is to allow more people to use the system and let them determine its benefits. The current system can be used for many purposes. It can help users understand arguments by laying out the individual components according to their relations, or it can be used to construct new arguments by using the display to direct their thoughts. Students have used the system to organize ideas when answering homework questions with good results which suggests that the system could be very useful in academia. We are optimistic that using EUCLID to analyze and develop reasoned arguments will lead us toward better understanding of these common yet difficult tasks.

Acknowledgements

This research has been supported by NSF grant IST-8609599, a grant from Symbolics, Inc., and by the Department of Computer Science and Institute of Cognitive Science at the University of Colorado at Boulder.

References

1. Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer and Michael Woolf. Constraint Hierarchies. OOPSLA, 1987, pp. 48-60.

2. Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces". *ACM Transactions on Graphics 5*, 4 (Oct. 1986), 345-374.

3. David Harel. "On Visual Formalisms". *Communications of the ACM 31*, 5 (May 1988), 514-530.

4. John B. Smith and Stephen F. Weiss. "An Overview of Hypertext". *Communications of the ACM 31*, 7 (July 1988), 816-819.

5. Paul Smolensky, Barbara Fox, Roger King and Clayton Lewis. Computer-Aided Reasoned Discourse or, How to Argue with a Computer. In Raymonde Guindon, Ed., *Cognitive Science and its Implications for Human-Computer Interaction*, Lawrence Erlbaum Associates, 1988.

6. Zukerman, I. and Pearl, J. Tutorial Dialogs and Meta-Technical Utterances. Computer Science Department, UCLA, 1985.