Euclid: Supporting Collaborative Argumentation with Hypertext

Bernard Bernstein

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

Abstract

Many books and papers are written collaboratively across great distances and over long periods of time. Lately, research has been focusing on real–time collaboration, but little has been done to provide a means to share reasoning like that used in research collaboration. Euclid is a collaborative hypertext system for creating and analyzing reasoned discourse over large spans of distance and time. A Euclid argument may represent a complete argument from a single individual or from many participants with any number of perspectives. As a writing tool, Euclid allows users to fully analyze the logical structure of their arguments to create a clear case when putting it on paper.

1. Introduction

In the past few years, researchers have been studying schemes to represent reasoned discourse. Traditionally, arguments are represented on the printed page sequentially. In the 1950's, Toulmin suggested a graphical representation for argumentation which follows a strict convention (Toulmin, 1958). In recent times, researchers have been using computers to implement Toulmin and other graphical representations. The latest systems provide practical methods for manipulating discourse. This paper discusses a highly usable tool for collaboratively developing and analyzing reasoned discourse with a user–centered philosophy.

The Euclid project addresses two major areas of modern computer science: hypertext and computer supported collaborative work. It is a hypertext system because it supports network–based structuring of nodes. Links between nodes may be traversed to discern the structure of the network. As a collaborative application, it allows several users to work together, share ideas, and create networks. Users contribute to the overall structure as individuals or as part of a group.

One can define Euclid as a visualization tool. It provides a framework in which to represent reasoning *visually*. Visualization tools represent complex concepts in such a way that the complexity is hidden by a metaphor which is easier to comprehend. Euclid allows users to create arguments visually, constructing branches of an argument that are displayed like branches on a tree. These branches can be climbed to search for meanings within the structure.

A figure in a paper, which is a two dimensional entity, can usually clarify a concept significantly faster than describing the same concept in words. Authors insert pictures into documents to illustrate ideas because "a picture is worth a thousand words." Similarly, a picture of an argument is valuable for the comprehension of its content.

1.1. Hypertext Defined

Traditional writing is sequential (Nielsen, 1990). Each sentence is succeeded by at most one other sentence in a linear structure. We are accustomed to this one–dimensional format, but we are also very restricted by it. The order of the sequential document is posed by the author for a specific audience. Since any member of the reading audience may have different background knowledge than any other member, some readers may have a better understanding of the document than others. A document which does not use a sequential ordering, one which may be read in various orders, could appeal to its audience at the level of each individual.

Hypertext is the concept of ordering text nonsequentially. A hypertext document is organized in multiple dimensions. When reading a hypertext document, an individual reads a piece of text, and then may choose several directions in which to continue to read. The Euclid system provides authors and their audiences a nonsequential platform in which to write and read reasoned discourse.

1.2. Long–term Collaboration

There are several levels of collaboration. Real–time collaboration is for tightly coupled tasks, such as making a business decision between employees (Schrage, 1990). Another type of collaboration is long–term collaboration, which is used for tasks which are done primarily by independent people, with checkpoints when they collaborate briefly to discuss their results. In academia, or the research community, most work is done by individuals, and then the ideas are shared with the rest of the community. The other researchers then use that information to help build their independent cases.

When researchers with opposing views get together, they may discuss how their works differ and offer opinions based on the published research done by each. The Euclid system supports this style of collaboration by allowing users to work independently and then build upon the argument over long periods of time.

A group of researchers may be working on their arguments for months, and when they meet or when they communicate through their computers, they can *merge* their arguments effortlessly using Euclid. The system allows participants to work independently to construct their own ideas while sharing their work in the long–term process of academic research.

1.3. Euclid

This paper is mostly a linearized version of an argument which was developed using Euclid. The argument will try to gain adherence of various claims which essentially state that the Euclid system is useful and could eventually influence how people develop arguments in the future.

Since this paper is written sequentially, we needed to create an ordering which attempts to make the argument readable to at least some of the audience. Section 2 presents some background information about the underlying principles that Euclid uses. Next, section 3 talks about the design of the program and issues that were involved with the design. Section 4 discusses how the application can be used for practical purposes. That section also contains examples of some arguments which were created and analyzed using Euclid. Finally, section 5 addresses some future directions for the project.

2. Concepts

There are many concepts which Euclid addresses, some of which are discussed here. The entire system concerns augmenting the process of working with *reasoned discourse*, so we discuss reasoning and its relation to Euclid. This section then gives some more details regarding the use of hypertext and collaboration and their significance to the Euclid system.

Some features of the Euclid program are discussed because they are concepts which need to be defined. The *typing* of objects, the definition of *sources* of the information in the objects, and the three primary data types are discussed in some detail here.

Finally, this section addresses the two document types which contain all of the data. The *database*, which stores the objects and their contents, and the *display*, which presents the objects to users are described and discussed.

2.1. About Reasoning

The primary goal of argumentation is adherence (Perelman & Olbrecht–Tyteca, 1971). An author of reasoned discourse is always attempting to convince his audience of some conclusion. There are many different styles of persuasion used by authors. Appeals to emotions, confusion, analogies, and deception are a few techniques writers use (Rieke & Sillars, 1984; Walton, 1989). Many of the techniques do not follow any logical foundation even though the author may attempt to convince his audience that the argument is logical.

In its purest form, a logical argument is like a mathematical proof. In a proof, statements, called theorems, are presented which have been proven to be true. A proof uses logical deduction to ensure that no new claim is made unless all supporting statements are true and they conclude the new one.

The conclusion of a proof is undeniable if the statements themselves are true and the conclusion logically follows them.

In a natural language argument, a presenter attempts to prove a conclusion by making statements that prove its validity. If the argument is proven by *modus ponens*, or logical deduction more generally, then the presenter stands to gain adherence from the audience. If the argument is unclear or does not provide support for its statements, then the audience may be more likely to doubt its conclusion. Many writers substitute strong rhetoric when the argument lacks logic. Other writers may have a solid logical argument but are not eloquent enough to present it well. A critical analysis of arguments from these two styles would show that the logical argument is more substantial despite its expressive shortcomings.

Some argue that strong rhetorical arguments are as valid as strong logical ones. We are influenced greatly by words because they can have subtle psychological meanings and conjure deep emotions. Some rhetorical arguments may have more influence on their audience than good logical ones, but once the logical structure has been exposed, the logical one will continue to influence.

If an author writes an argument using Euclid, the logic will be apparent to the reader. If the author does not have a conclusive logical argument, then writing it with this system will help the author see this lack of logic as it supports the process of making the argument more sound.

When a reader is trying to understand a written argument, she may not be able to follow the logical structure. If she imports the argument into Euclid, the logical structure can become visible in the multi–dimensional realm of hypertext. The process of importing the argument can also give insights into the organization of the document.

Arguments which use strong rhetoric or bold claims are good candidates for Euclid analysis. Often the bold claims are based on unproven assumptions. When analyzing an argument using Euclid, it becomes obvious to the reader when this trap is made.

2.2. Hypertext

Many research and commercial systems address the defining concepts of hypertext systems. All of these systems implement atomic units which are independent of each other. These units, which we will call *nodes*, contain some representation of information or knowledge. Connections between these nodes give the audience a method for traversing them nonsequentially. *Links* may connect nodes with each other in any order, and the author of the document may assign the order arbitrarily. From this point on, the various hypertext system designs differ in some way.

Each hypertext system has its own method for representing the nodes and connections, and many have other fundamental data types. Hypercard, originally from Apple Computer, Inc., uses the *card* as its node. A card, which contains text, pictures and sounds, may be connected to any other card by way of a link assigned by the author. A Link in Hypercard is represented as a *go to* instruction in its scripting language (Apple Computer, 1987).

By virtue of this form of linking, cards may only be traversed by leaving the context of one card and entering another. This form of data traversal appears to the user as a *mouse–eye* view of the data. The metaphor is that of a mouse running through a maze where its only perspective is of the walls currently surrounding it. In Hypercard, the user is only able to see the data from the context of the current card. Figure 1 illustrates the mouse–eye representation used by Hypercard.

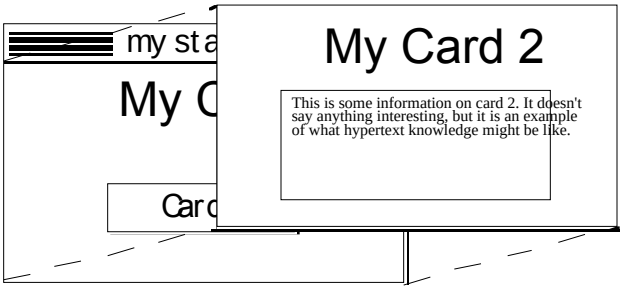


Figure 1. Hypercard navigation. The user clicks on a button in one card and another card replaces the original.

In contrast to the Hypercard style of navigation, some hypertext systems, such as gIBIS (Conklin & Begeman, 1988), allows users to see the structure of the database through a *birdseye* view. In this representation, the data may be viewed from the context of the entire database. Several nodes may be visible from the perspective of a bird hovering over the database. Figure 2 shows a representation of the same two Hypercard cards from a birdseye perspective.

In NoteCards, a hypertext system developed at Xerox PARC, both of these representations may be used (Halasz, 1988). When the user is looking at a card, connections on the card may be used to display other cards. The NoteCards system also provides an overview mode of the nodes, called a browser, which displays a spatial layout of the cards. This spatial layout of nodes may be used to organize data in an additional dimension; that of visual orientation.

It has been observed that all of these organizations are useful for different tasks (Marshall, Halasz, Rogers, & Jannsen, 1991). If each node contains large amounts of data with structure consisting of simple relationships, then the mouse–eye view is advantageous. A user manual, for example may have a complete description of a program feature within a single node. The links may be used to connect related commands to the one in the current context. As a programmatic description of a system, this is practical. On the other hand, most academic, journalistic and other written documents are not structured this way.

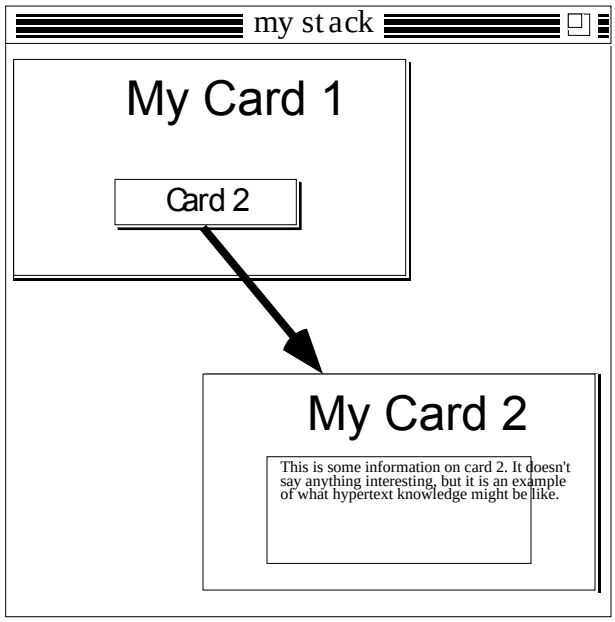


Figure 2. If Hypercard had a birdseye perspective, it might look like this. This type of perspective is useful to browse large information stores where the structure is important.

In academic writing, a thesis is proposed and arguments are written to support it. The overall structure of the information in the argument is of great importance to its validity and ultimate adherence. For this type of argumentation, it is necessary to support the visualization of the structure from an overview perspective. Euclid Supports this style of hypertext interaction because it is the most appropriate for aiding in the development and understanding of reasoned discourse.

2.3. All Objects Have Types

When a user is creating an argument in Euclid, he assigns a *type* to every object. The three fundamental types are *Text Objects*, *Relations* and *Lists*. All objects are based on one of these types. Users define types in a hierarchy; each type is associated with a set of *parent* types. Figure 3 shows part of a typical type hierarchy for each of the three basic types.

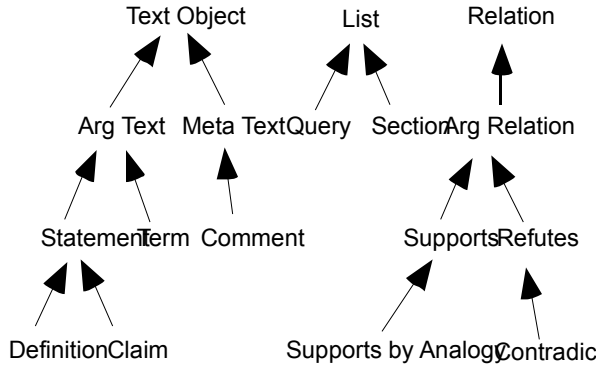


Figure 3. A typical type hierarchy. The arrows connect each type with its parent type. The top row contains the fundamental types.

Definable types give users an extensible system capable of representing any style of argumentation. When an author introduces a new object which does not match any of the existing types, then he may define a new one to accommodate it. An author may even create *meta–objects*; objects that are *about* the argument and are not *part* of the argument.

Hierarchical types may be used for more or less specificity of objects. Very specifically classified objects are ones with types that are deep in the hierarchy and less specific types are near the top. If an author creates an object that fits well into a very specific class of objects, then the type for a deeply defined class can be used. If, on the other hand, the author creates an object that does not categorize as well, then a more primitive type can be used.

Users use these types to follow the logic of an argument. A reader can see clearly when a *claim* is *supporting* another claim or when a *comment* is *about* a definition. In addition to the user's perspective of the types, the program uses them as its semantic interface to the argument.

Since the Euclid system does not implement natural language understanding of the text objects, the only information it has about the argument is that of the structure. The user's interface to the computer's representation of the structure is through a query. The query takes advantage of the hierarchy by searching

for types or sub–types.

2.4. The Source of Knowledge

An integral part of Euclid is the representation of *sources*. Every object in the system contains a source, which is the person or perspective from which the object was made. In many cases, the source is the user who is writing the argument. This is not always the case.

Often writers make claims which are not from their own perspective. For example, an author can write a claim which she is opposed to in order to build a case for why she is opposed to that counter–claim. She can define that other source as the "anti–me", to represent general opposition to her own ideas presented.

Composite sources can be created to represent sources

within sources. For instance, “Franklin said that Jefferson said…” would represent something that Jefferson did not necessarily say himself, but rather something that Franklin said that Jefferson said.

When an argument is created collaboratively, let’s say by two opposed authors, the first set of claims by each side may contain only objects with themselves as the source, but when they see each other’s claims, they may begin to make claims which paraphrase each other. A paraphrase of writing by another author could use that other person as the source even though the creator of the object was the opposing author.

Sources give readers a more complete perspective on the arguments. They allow the reader to follow some of the more intricate connections which authors make.

2.5. Nodes Contain Content

In Euclid, as in other hypertext systems, a node contains knowledge. In the current implementation, only styled text is used. Future versions should be able to have any representation of content such as graphics, sound or animation. In addition to the content, type, and source, the node also remembers the user who created it, the time it was created and the time it was last modified. A name may be associated with every node as a label for quick identification. Figure 4 shows what a text node looks like in the display.

Figure 4. This is what a text object looks like. The top bar contains the type; it can also contain abbreviations for the creator and the source of the claim. The second bar is a brief title for the object. The rest of the object contains its content.

The information in a node may only be modified by its creator. Some have suggested that the system allow any user to edit nodes, or to provide access privileges to various users, but that is not feasible due to the nature of long–term collaboration. When several users are editing a document with their own copies of a database, if multiple users change a single node, then there is no precise way to reconstruct the complete database with consistent data. The style of collaboration which Euclid supports allows for participants to communicate through mail, e-mail or computer networks, so there may be no connection between the machines directly to implement revision control.

Nodes created by the current user are called *native* nodes and those made by other users are called *aliens*. Even though only one user may edit the contents of a node, any user may manipulate the node as a whole. Users may look at alien contents and make connections between them regardless of who created them. Users can extend arguments by displaying and moving alien objects, and adding native ones to connect to them through native relations.

If a user disagrees with the structure of an argument, or wants to reconstruct the argument with a different layout, that user may create a display and connect the nodes differently than the original author. The content of the alien nodes do not need to be changed to do this.

Since the nodes are given types, they may hold different types of information. One type of node may contain *meta–information* which refer to various pieces of the argument. This are analogous to post–it notes, but they are more powerful than simple snippets of paper. Complete meta–arguments can be created which discuss pieces of the argument structure.

2.6. Relations Represent Structure

Relations share many properties of nodes. They, too, are typed, have a creator, creation and modification date and an optional name. Relations do not contain content, but they do contain links. Relations connect objects to each other and provide a type to the connection. They *own* links, which means they are used like a node which connects several other nodes that give semantics to the connections.

By connecting two objects, the user is making a logical connection between them. These connections are not limited to text objects. Relations may be made between other relations, text objects or list objects. For example, one user may disagree about the structure that another user made. That user may make a refutation relation between her claim and the supporting relation.

In figure 5, the relation has boxes at the ends of the lines to represent the ownership of the lines. In many cases this representation is redundant, but if the line is connecting two relations, then without these boxes the meaning of the line would

be ambiguous. For example, in figure 6A, X *supports* Y and Z *refutes* the *supports* relation as well as another object, W. Figure 6B shows the exact same set of objects and connections, but in this case the fact that Z *refutes* W helps to *support* Y. The owner of the line between *refutes* and *supports* changed and modified the meaning of the argument.

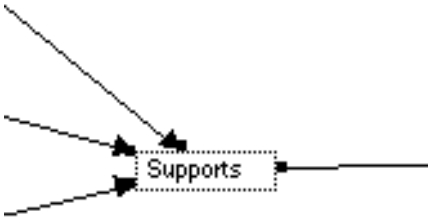


Figure 5. A relation object contains information about its connections. The small boxes at the ends of the lines in the figure represent that this relation *owns* these lines.

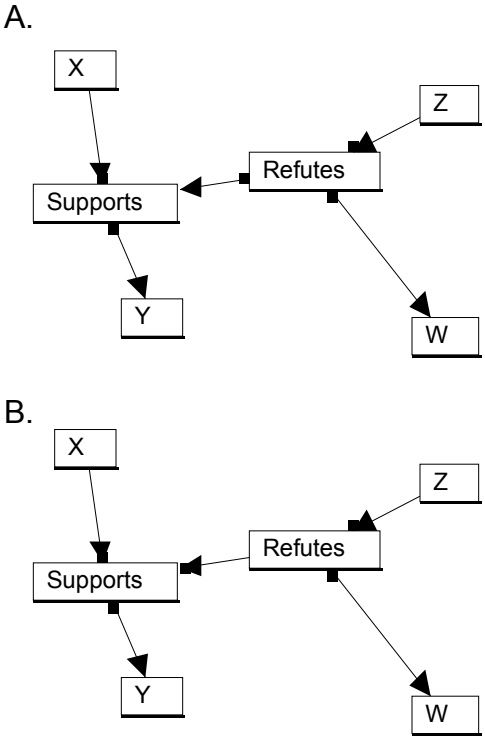


Figure 6. A: Z refutes W **and** the relation from X to Y. B: Z refutes W, and this refutation helps support Y.

2.7. Lists Create Complex Relations

Like relations, lists share many properties of nodes. All the same attributes are associated with lists, but where nodes contain text content and relations contain links, lists contain ordered sets of other objects.

Figure 7. A list object contains an ordered set of pointers to objects.

A list object is used to group a set of objects. One type of list object may group a set of claims made in a written document, in the order in which they appear in the paper. Lists are ordered sets of objects which associate the objects to each other in some way through its type. The members of lists are not limited to nodes and may contain relations, text objects or other lists. Figure 7 illustrates a list object in the display.

Several operations may be performed on lists and we are continually finding very useful applications for them. Lists may be created by selecting a set of objects and then creating a list which contains them. The order of the list can be changed by the user by direct manipulation. This pair of operations allows users to maintain a linear form of the argument within the hypertext document. This linear form of the argument can be manipulated as a whole, or pieces of it can be revealed by selecting portions of the list.

In the sense of linearization, the list object can act as an agent between the logical layout and the sequential version of the argument. A user can invoke the *copy* command on a list, and then *paste* the list into a text editor. This operation exports the text of the list members from Euclid into other programs in the order of the objects in the list.

Lists provide a means of representing a portion of an argument in a single structure. A set of objects can be transformed into a list object. Other objects can be related to the list in the same way that objects are related to each other. This method can be used to support a section of argument, or for a conjunction of objects to support another claim.

The query operation creates a list containing its results. A user can perform a query which requests all objects created by a particular user, for example, and receive a list of the matching objects. Once a list is present, further queries can be performed on it to find a specific subset. A second query may solicit objects from the first list that match additional criteria. The query options are shown in figure 8.

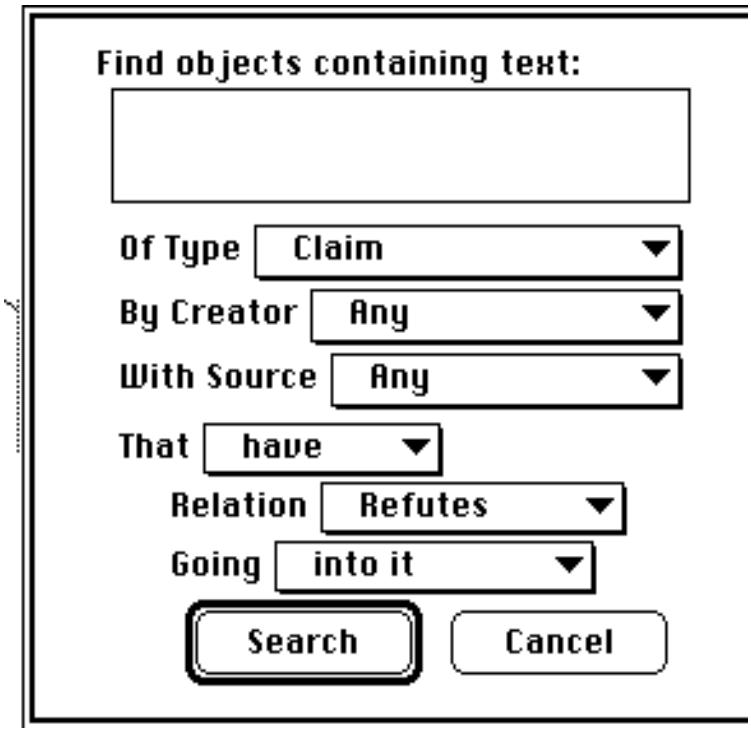


Figure 8. A query finds any objects in the database that match various attributes.

We first introduced list objects in the current version of the system, so we are still developing useful applications for them. We believe they are a very powerful data construct which can be used in many different situations.

2.8. The Database Stores Everything

Implicit in all hypertext systems is the concept of a database. The database stores the knowledge that is accessible from the displays. In Euclid, the database is an independent system within the environment. All of the data, types, users and other information is stored in the database and available for quick access. The only information that the database lacks is display information for objects.

Location, size and other display flags for objects are stored in the displays, not in the database. This is a vital separation for this system. This separation of data and layout permits multiple views of the data in separate windows simultaneously. It also helps support the collaborative aspect of the system.

Euclid supports collaboration in three ways. First, the content of every object and type is *locked* so that only the creator can modify or delete it. Second, any time the content of an object or the definition of a type are changed, the database stamps the time on the object. Third, databases may be merged into each other.

When a database is merged into another, the latest versions of every object replace their earlier versions. This allows users to merge the latest version of their database with an earlier database that was extended by another user. Since the database does not contain any display information, the same display files will continue to work with the database, and objects which have new relatives can be traversed to find the objects added by the other user.

Euclid provides users with a database browser so that the entire database may be viewed without opening a display. This browser is not intended to be used for reading or writing arguments, it simply provides a crude method for accessing all objects in the database. Future versions of the program may have a more useful front–end for the database.

2.9. Displays Are Layout Editors

The display is the primary editor and viewer for the system. A display contains a graphical representation of a subset of the database. Displays contain pointers to objects in the database with the addition of layout information for each object. Since displays depend on a database, they can only be used when the matching database is open.

Since the database stores the information autonomously, several displays may be created which show various sets of objects from the database. These displays automatically update their objects when they are changed on different displays. Moving and resizing of objects on one display does not affect any other display. Many other operations may be performed on objects in a display which **do** affect the database, and therefore the other displays.

If a user *deletes* an object from the database, then any representation of that object is deleted from all displays. Any lines connected to that object are also deleted. Alternatively, objects may be *hidden* from a display. This does not affect any other display, but it does remove the selected objects from the current display. When text objects are edited in a display, they will be revised in the database and on all other displays that show the objects.

Users may move objects between displays by *copying* them from one and *pasting* them into another. A user can also copy and paste from the database browser to a display. There are several other ways of introducing hidden objects into displays.

A user can perform a query which creates a list object on the display; then the list object can be asked to show some or all of its members. Figure 8 contains the window that is used to form a query.

If a visible object is related to hidden objects, the user can traverse its links to display its hidden neighbors. See figure 9 for an illustration of a hypertext traversal.

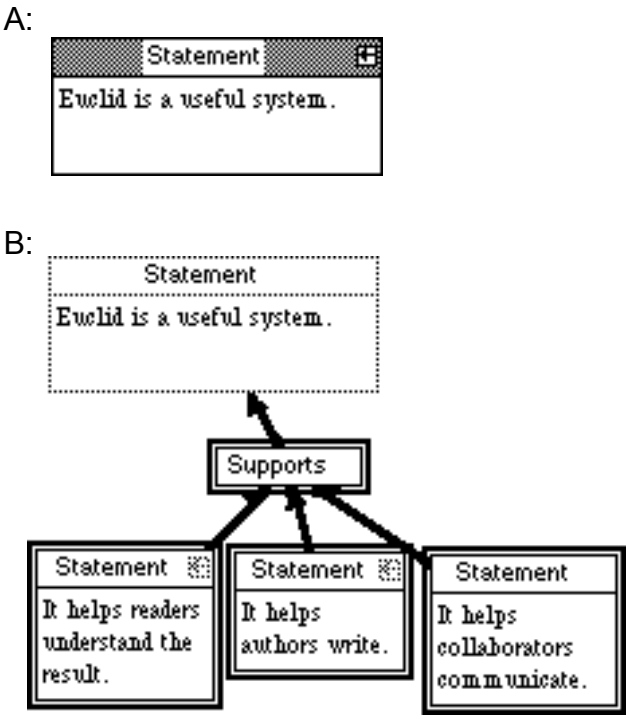


Figure 9. A: The left–arrow icon on this node represents hidden connections to it. B: The relatives of A were exposed by clicking on the link icon.

Each of the concepts discussed in this section contribute to the design of the Euclid system. The ideas from the area of argumentation helped shape the design by providing background information about how reasoning works. Hypertext systems influenced the design of Euclid in several ways.

Hypertext supplied the basic idea of viewing the text nonsequentially. It was also the inspiration for the node/link model and the creation of separate and distinct database and display systems.

We developed the typing scheme as a method for enabling an extensible system in which various forms of argumentation can be achieved. The source information for objects gives users a context in which to help understand the background of objects.

3. Euclid Design

There are many components to the design of the Euclid system. The complete design of the system is beyond the scope of this paper, but we will discuss some of the issues which were addressed during the design process.

In addition to the concepts discussed in section 2, we will discuss the design issues associated with developing a collaborative system.

The program was written with an object–oriented approach using a library of classes which handle some of the general features of the user interface. Euclid implements many subclasses from the library and also defines some new abstract classes. The communication between the various components of the system are discussed here as well so that the reader can begin to understand some of the issues involved with connecting independent components of a large system.

3.1. Collaborative Design Issues

When developing this system, it was originally intended to be used only by a single user. Eventually, we discovered that its usefulness grew beyond that of a single–user application and was ideal for collaboration. Several problems appeared when we attempted to structure the database to allow for collaborative operations. Problems arose in providing unique identifiers, object consistency across remote databases, and type consistency.

3.1.1. Unique Identifiers

The identifier of objects needed to remain unique across unconnected machines. It is impossible for a machine to produce an identifier and guarantee that no other machine has made the same one without communicating with the others. We can, however, make an identifier that is unlikely to have been replicated on another machine.

A random 32–bit number has over 4 billion possibilities. If two instances of the program make random numbers, the chances of their being the same is one in 4 billion. If a thousand entities are created independently with 32–bit identifiers, the chances are approximately one in 10 thousand that two of the numbers will be the same. This seemed improbable enough for objects which are not numerous in the system. Types, users and sources are given a 32–bit identifier because there are few of each of these entities.

Databases can grow rather large, containing thousands of objects. To create unique identifiers for instances of database objects, the creator's identifier is concatenated to a new random 32–bit number. When a user creates a new object, a new identifier is issued which has not already been issued to that user. This uniqueness can be verified within the same machine assuming that only one instance of each user exists. This guarantees that no two objects can have the same identifier as long as no two users have the same identifier.

Of course the seeding of the random is the weakest link. The current implementation seeds the random number generator with the time that the program was launched. This method may be improved for future version.

3.1.2. Object and Type Consistency

The problem of object consistency is addressed by stamping every object with their latest modification time. When a database is merged into another, all objects which already exist are compared with the new ones being merged in. The newer of the two replaces the older one.

If a user deletes an object from a database that has already been merged with another database, then the object will reappear when the database is merged back. In other words, once a user releases his claims to other users, they can never be deleted. The best he can do is make another statement which withdraws his previous claim. This is analogous to an authors publication of a paper which she later realizes is not valid. She may not take every copy of the paper away from her readers and pretend that it was never written. She can, however make a public statement correcting the error.

A design feature which may not be completely acceptable is the ability to modify claims. A user may make a claim and then change its content, and thus its meaning. It behooves authors to be sure that editing of objects do not change their meanings after other users have exchanged databases. Another user may make statements and connections which depend on the meaning of a claim, but if the original author changes the text of the statement, then the other user may have a meaningless argument.

A similar problem is possible with types. When a user defines a type, other users may use that type to instantiate their own objects. If a user defines *supports* and later changes the name of the relation to *refutes*, then the entire meaning of the database can change.

One way to correct these problems would be to restrict editing after the database has been distributed. This would eliminate the problem of changing of meanings for objects and types once other users have seen the database. The primary problem with this approach, however, is that typographical and other minor errors could not be corrected. The other difficulty would be in knowing when the user distributed the database. As it stands now, a database can be transported at any time as a file without any special operation.

3.2. Object-Oriented Design

Euclid was written for the Macintosh using Think C by Symantec, a C implementation with object extensions. The program makes use of the Think Class Library, a library of code which implements many of the mundane, low–level functionality of a Macintosh program. The specific code for Euclid takes up more than 55 classes of objects and more than 800K of source code.

Since the program is behaviorally object–oriented, it seemed natural to implement it with an object–oriented approach. Using objects as abstractions for the various entities made the programming process a smooth task. The data encapsulation of the objects eased the implementation of the separate data containers.

Two abstract classes that were defined for the database system include a *hash table* and a *hash table member*. A hash table implements a data structure which can access any of its elements in almost constant time, provided that all objects have unique identifiers. A hash table member is a class which stores a unique identifier, and, as such, can be stored in a hash table. The unique identifiers which the program use are the 32 and 64–bit numbers discussed in section 3.1.1. All object stores in Euclid are subclasses of the hash table. The main database, the display database, the type database and the user/source database all depend on hash tables.

The implementation of line manipulation is an issue which is not well documented in Macintosh programming literature. It is easier to implement objects of almost any shape as a region, but since lines do not enclose a region, there are no primitives to operate on them. The Euclid code contains an object definition for handling lines. A line basically consists of two endpoints. More specific lines can connect two objects and have an arrow at one end. A method for the line class determines if a point is near the line so the program can decide if the mouse cursor is on it.

3.3. Communication Between Objects

The *Database* contains all *Database Objects* and the *Display Databases* contain the list of all *Display Objects* for any given display document. Every Display Database is dependent on a Database. The program supports a method for these various components to communicate.

The Think Class Library provides a framework for dependencies between objects. One can define an object of one class to *depend on* another class. When an object performs an operation, it can *broadcast* the event to its dependents. It sends a message to all its dependents, describing what changed and how it changed.

These dependency links can be created between almost any pair of objects. One could create a dependency between every display object and its respective database object, but the memory overhead associated with having large numbers of dependencies would be too great. For this reason, the dependencies between the objects are controlled by their containing objects: the database and the display, not the individual objects.

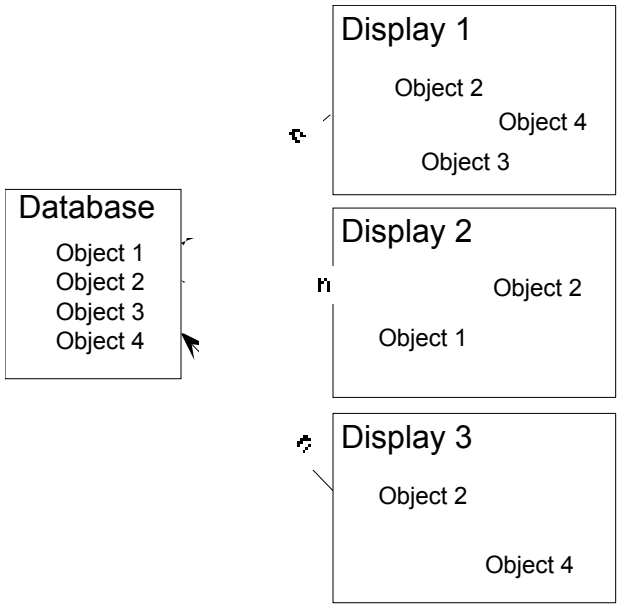


Figure 10. The displays are dependent on the database. When the database changes an object, it broadcasts a message to all dependents. The displays receive the message and update their displays accordingly.

Figure 10 helps illustrate an example of a typical transaction between the display and the database. There are four types of objects represented in the figure: a Database, three Display Databases, seven Display Objects, and four Database Objects. There are actually several other objects involved with this scenario, but we will simplify the transaction to the ones in the illustration.

A Display Object receives some action from the user. It sends a message to its Display Database with its object identifier. The Display Database passes another message to the Database with the same object identifier. The Database performs the task on the Database Object and Broadcasts the change. Each of the displays are Dependents of the Database, so they receive the message that the Database broadcasted. Each Display Database then updates its objects accordingly.

3.4. Design Evolution

The Euclid system took several years to evolve,and it was not originally designed in its current form. The first incarnations of the system were written in Lisp on the Symbolics Lisp workstation using Flavors, a predecessor to the Common Lisp Object System (CLOS). In the past three years, it has been redesigned and implemented for the Macintosh.

The Symbolics versions had two components that were omitted

from the Macintosh version: a constraint satisfier (Smolensky, Bell, Fox, King, & Lewis, 1987a) and an Argument Representation Language (ARL) (Smolensky, Fox, King, & Lewis, 1987b).

The constraint system was used for distributing the layout of the objects based on their relationships. This was a very processor–intensive task and was eliminated in favor or direct user manipulation of the objects. This is discussed further in section 4.2.1.

The program used ARL, the Argument Representation Language, as a formal language to express the argument. An ARL expression consists of a section of the argument. *Supports(A, B)* would be an expression which is represented as object A supporting B, or graphically as two objects with a *supports* relation between them. The current version of the program implements a database which is analogous to ARL, and can express most of the same structures, but it is not represented in a formal language. Since ARL is a formal language, it is able to express more complex structures.

In ARL, sources, claims, and expressions were manipulated and passed as parameters to complex relations. A complex relation can define composite structures such as:

```
strong_misrepresentation(X, Y, cl) :=
        asserts(X, asserts(Y, cl)) &
        asserts(Y, not(cl))
```

In this example, a ternary relation is defined between two sources and a claim. If source X strongly misrepresents source Y, then he asserts that Y asserts a claim, but Y actually asserts some other claim.

In the Lisp implementation, ARL was practical and useful, but in the C version, supporting ARL would have been a very difficult task. Perhaps future versions of the program can include ARL as a formal representation of the arguments.

4. Using Euclid

This section gives some suggestions for using Euclid. The system may be used for a broad range of applications, but here we discuss the applications of the program that address the primary goals of the project. Every researcher must read and write reasoned discourse as part of her work. Conference papers, theses, and technical reports are all forms of reasoning. Some of these papers are written alone, others are written by groups of people. Euclid supports both ways of writing arguments and it helps an individual understand the discourse written by various authors.

4.1. Reading and Analyzing Argumentation

4.1.1. Structuring a Written Argument

When analyzing a written document, the user may use any method that makes the task easy and productive. This section discusses some approaches to argument analysis that Euclid supports well.

Some research has suggested that the Toulmin structure (Toulmin, 1958) can clarify logical structure. This structure is a standardized set of conventions for representing arguments formally (Jarczyk, Löffler, & Shipman, 1992). Using Toulmin structure, the reader follows a standard structure which divides the argument into a specific set of components.

Figure 11 illustrates a small segment of an argument using the Toulmin structure in Euclid. The Toulmin structure forces all arguments to fit into this type of structure. A *datum* supports a *claim*, and the *warrant* supports the connection between the two. Additional elements may be entered as well. For example, *backing* can support the datum, and a *rebuttal* may refute the claim (Rieke & Sillars, 1984).

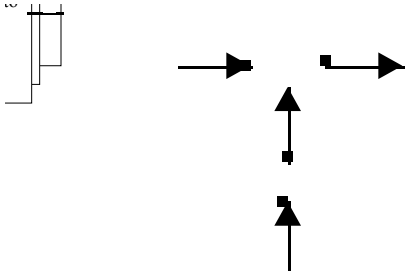


Figure 11. Toulmin has a well–defined set of conventions for representing arguments.

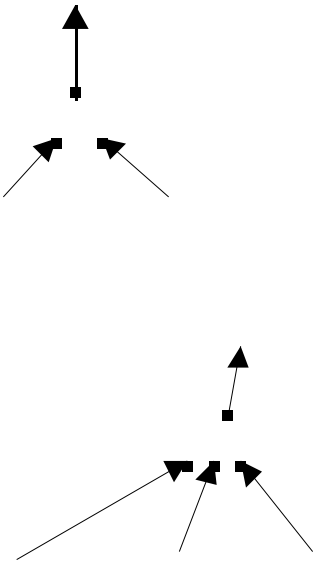


Figure 12. Euclid permits informal representations of structure.

Using Toulmin for argument structure is analogous to sentence diagramming. Some components of a sentence are its subject,

object and verb. Using this structure to break down sentences allows the reader to verify that the sentence follows the standard structure. The Toulmin structure forces a certain level of granularity to the argument. Each statement must satisfy the conditions for its role. The role of the warrant is that of a predicate which, when satisfied by the datum, establishes the claim. The warrant must be contained in a single cell, limiting the granularity of the cell.

The task of representing every component of an argument in the Toulmin form can be very tedious. When the argument is completely represented using this structure, it may also be lacking various pieces of the original document (Marshall, et al., 1991). Elaborations of the objects can be omitted, meta–comments about the structure might not be represented, and various other statements that do not directly lead to the conclusions will be missing.

Since Toulmin's formalized structure is limited by its granularity and its rigidity, an extensible system, which allows less formal connections is necessary. Euclid allows the definition of various types to accommodate the Toulmin style, but is also able to represent informal relationships.

A less formal representation of the argument segment in figure 11 is shown in figure 12. In this illustration, there is an additional supporting claim and an elaboration of that claim. When a user is performing an analysis with Euclid, this informal structure can make evolutionary changes. The analyzer might have the impression that one structure is logical, but then decide that a different structure would be more appropriate. This flexibility is advantageous when working with a large document.

As the reader is scanning a document, a high–level representation of the argument might be created, but when the contents of the individual sections are understood, the overall structure might change. Using Euclid, the user can make this change readily.

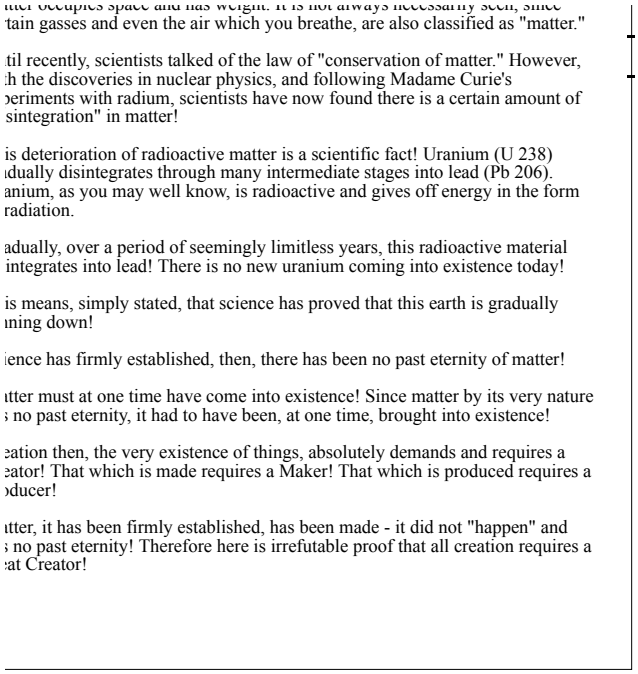


Figure 13. A small, complete argument.[1]

4.1.2. How to Perform an Analysis

When a reader wants to analyze a written document, he needs to import it into the Euclid system. Once it is in the Euclid format, he can manipulate and arrange the claims so that the implicit logic becomes explicit. He can follow several steps to transform the sequential claims into a network of claims.

The user can begin by creating a statement or defining a type, *quote*, which contains the exact text of the original document. The *source* of the object needs to be the source of the document; either the person who wrote it or a reference to the document will suffice. The first quote will contain the first statement in the document. If the first sentence makes a single statement, then **it** can be used as the first claim in the argument. If the first phrase of the first sentence or the entire first paragraph make a single claim, then **they** can be used. The user can enter the text either by copying it from another file and pasting it into the text objects, or he can type it in.

[1]From Bible Study instructional material published by Ambassador College (Pasadena, California)

After a few claims are entered, or when the entire document is entered, the user may begin the analysis of the argument. This process is similar to the brainstorming process that writers use. They dump many ideas and then try to make a logical argument out of them. In an argument that already exists, the reader attempts to perform a similar task with the author's ideas.

In the following example, we perform an analysis of a brief document.

Figure 13 contains the complete document which we will be analyzing. The first step in making an analysis is to import the document. In this figure, we simply copied the entire text and pasted it into a single text object in Euclid. This operation can also be performed by copying each claim individually and pasting them into their respective text objects. Having the entire text in a single object is only useful here so that the reader can see the full text in one place. The user of the system will still need to separate the statements for the analysis.

The next step in analyzing the document is to sort out the statements to find which ones are related to which others. The reader may hide some of the statements which are not useful for her level of analysis. The user

may move related claims near each other in the display and group them in order of logical flow. Once some of the connections become clear, she may connect the objects to each other using the relations that seem appropriate to the structure of the argument.

Once all the objects are connected to each other, an analysis may be performed. Figure 14 shows a complete representation of the argument. With this representation, the reader may look for flaws in the argument. Sometimes relations are made because the reader believes the author had intended for the connection to be made. When the connection is not obvious, the analyzer can add elaboration of the relations, describing why she thought it belonged there.

While the reader is following the flow of the argument, implicit assumptions may be found to complete the argument. The reader can make these assumptions explicit for future readers of the argument. In figure 15, the reader describes why a set of claims support another claim by stating an assumption.

The analyzer may continue to work with the argument by adding more supporting or refuting claims to any part of it. One might refute the assumption, for example, by questioning if the assumption is a valid one. A refutation of a claim may invalidate the claim and weaken the overall argument. Euclid will not decide for the readers if the arguments are valid or not, but Euclid can help the readers decide for themselves.

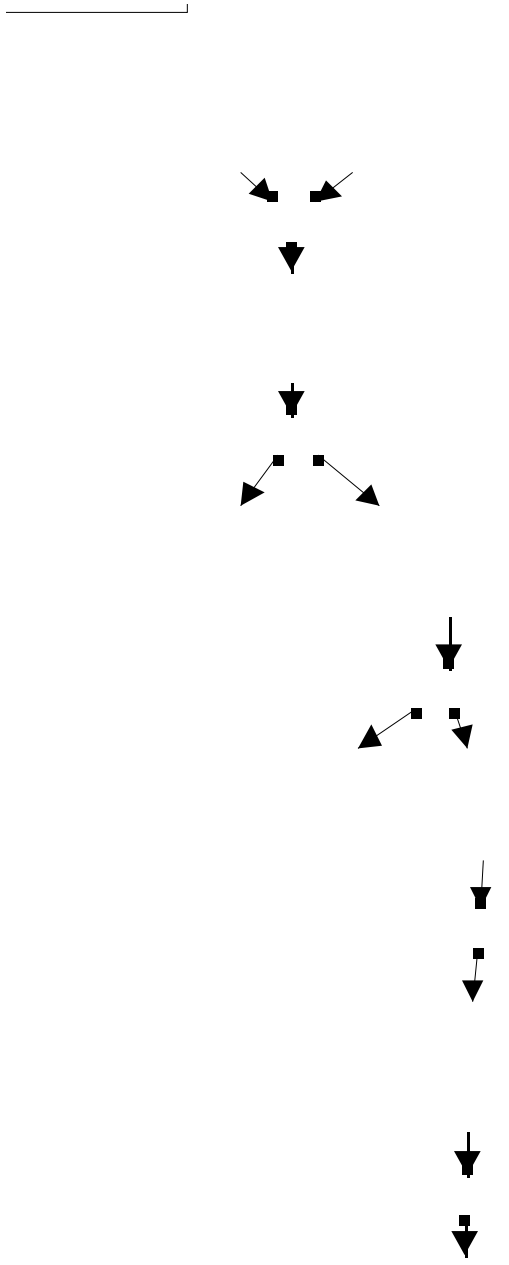


Figure 14. An analysis of the argument. This does not contain the entire text because the analyzer decided not to show the elaborations in this display.

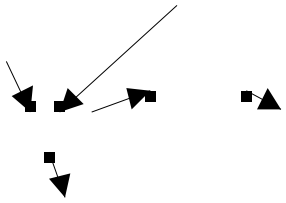


Figure 15. The analyzer adds an assumption which is not explicitly in the text.

4.2. Writing Arguments

When creating a document, users may follow any number of steps. Some authors begin their writing process by writing an outline, then for each section, a set of subsections are outlined. This process is repeated until the body of the paper develops from the outline.

Other writers begin by brainstorming sets of complete ideas. They may write complete paragraphs in random

order, and then sort them into a logical order, adding connecting sentences to complete the paper.

The Euclid system supports the writing process in every stage. It supports the initial brainstorming of ideas, then it augments transition from disjoint ideas into logical structure. When a complete argument is constructed, Euclid then helps the author transfer the logical structure into the linear, written form.

If the author prefers to outline and then specify, a similar process can be used. As ideas are created, they may be immediately moved near related objects in an outline form.

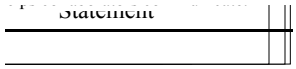
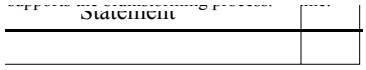


Figure 16. In this example, the user begins the brainstorming process by creating some random statements.

4.2.1. Brainstorming

When brainstorming, we are primarily concerned with expressing as many ideas as possible in as little time as possible. Our minds are filled with thoughts and we want to spill them out and record them so that we will not forget them. We are not concerned with making logical arguments at this stage, but we can follow our *train of thought* while ideas appear mentally.



At this point, it is very important to have complete control over the placement of the objects. We may think of several claims very quickly and have a vague idea of which other objects they relate to. Using Euclid, we can immediately move these thoughts near other, similar or otherwise related objects. New objects which have little relationship with the rest of the network can be placed in a vacant area.

Figure 17. Here, the author begins to group some of the ideas spatially while adding more statements.

Euclid supports brainstorming in two ways. First, it gives the user complete control over creation and layout of objects. Second, the program does not require explicit relations to be present at all times.

Users utilize direct manipulation to move the objects in the display. This unconstrained layout allows the use of spatial relations for the initial grouping of ideas. Unrelated concepts can be placed in different areas of the page or in separate displays.

In earlier versions of this project, Euclid had the feature of constraint–based layout. Whenever a relation was created between two objects, a supposedly useful spatial relationship

was created between the objects. For example, if X *supports* Y, the object X will appear just below object Y. When objects were not explicitly related, there was no support for automatic layout (Bernstein, Smolensky, & Bell, 1989; Smolensky, et al., 1987a).

Users of the earlier program complained that they preferred to place the objects where they wanted, and could usually find a better layout than the program could. In addition to the objections to constraint–based layout, the cost of satisfying the constraints was extremely high computationally, so this feature was abandoned.

This direct–manipulation approach to layout is important for the initial phase of quickly presenting thoughts into the system. Once a few of the thoughts are entered, the user may need to begin developing some higher–level semantics to the disparate ideas. Fortunately, the user is not forced to do that for each object while it is being created.

Some hypertext systems require explicit links between objects before their connections can be of any value. In Euclid, as in NoteCards and gIBIS, objects can be unrelated through the semantics of the program, yet still have a visual relationship by way of their proximity to each other.

Since the system does not constrain the user by demanding that the structure be present, he may brainstorm by creating disjoint ideas. These ideas can be organized visually in a display so that vaguely related ideas can be distributed near each other. This supports the beginning of the incremental transition from autonomous thoughts into more complex arguments.

The first stage of brainstorming is the dumping of ideas. In Figure 16, the user has created some statements which should eventually get related to each other or to other objects. At this point, the user is not concerned with the connections, but rather with the recording of the ideas.

In the next stage (figure 17), the user begins to group the disparate ideas so that they can be found when the relations are added. Some relations may be created while more ideas are still added to the database.

In this case, some more ideas were entered after some of the grouping was performed.

In the third stage of brainstorming (figure 18), the user specifies the links between the objects. The new objects were sorted out and connected to their appropriate sections in the argument. This stage is also the beginning of the structuring task.

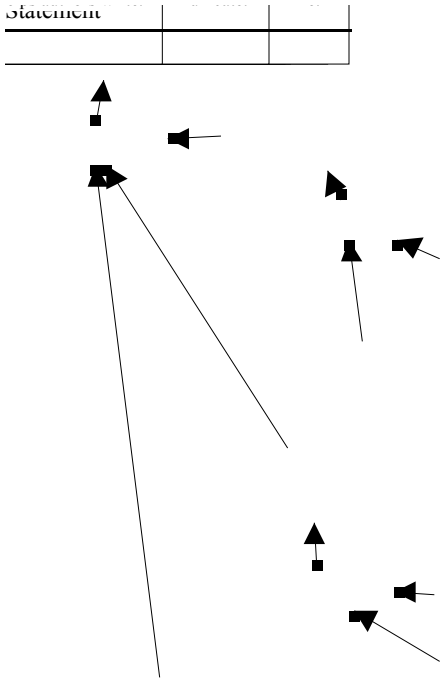


Figure 18. Finally, the writer connects the statements to each other.

4.2.2. Ideas into Structure

There is no tangible separation in time between brainstorming and structuring. The user is always brainstorming, in a sense, in order to add the details to the basic ideas. Since the structuring phase is at a different conceptual level, it will be discussed as a separate phase.

After some brainstorming, which may be done as the first step in creating an argument, or when adding details to another idea, the user will then begin the phase of structuring the argument. In adding structure to independent claims, users need support in at least two areas. Authors need a simple way to relate the autonomous objects, and they need to know where new objects need to be connected to the existing structure.

Creating Relations

Relations are supported as strongly as the text objects. They hold the same internal structure as the text objects, but instead of a text content, they contain connections. A relation is a node object which connects a set of other objects, the other objects may be text objects, other relations or list objects. Relations are extremely useful constructs when the user takes advantage of them.

Since relations have types, the objects may be connected in various ways. The author may connect objects with a *supports* relation, or define any other type such as *elaborates* for a statement which further explains another object.

After several statements have been entered into the system, the user may begin the process of creating explicit relations between them. This next phase of argument construction is not modal and is often done intermittently while the ideas are being constructed. While brainstorming, a user may create a few claims, and then group them spatially according to their topics. When the user has a sufficient group of related claims, she may see some obvious connections between them. Perhaps one claim supports another, or one is the definition of another.

At this point, the user will connect the related objects. A claim may be connected to another which it is supporting. Another statement may elaborate on a claim. Maybe a set of claims support a single statement. There may be a connection between two objects which the author does not yet know how to classify. Such connections may take on a generic relation which can be specified at a later time. If the type of connection is known but not defined, then the user may define it at that time.

As the user is creating connections, she may also find that some of the text objects may need more specific types. Something defined as a statement may be more appropriately defined as a claim or a definition. If the system does not have a type that is applicable, then the user may define new types. Types may be defined at any time or the user can always use a generic type until a more descriptive one can be established.

While the author is making connections, he might need to alter the layout of the objects. The author may find a spatial layout which reflects the logical structure better than the original, or he may want to accommodate the new relations with more space for the connecting lines.

Eventually, all the claims will be connected to each other, and a network will have been created. The user can now use the structure thus far to determine where to add more claims.

Adding More Claims

With a partially complete network, an author can find more information about the argument than she originally put in. The visual and internal representation of the argument can help the author determine how to continue to support the case. There are three methods the author can use to find areas which need more support. She may perform a query to find unsupported claims, she can look through the network to find *leaf* nodes, or she might look for claims which have weak support.

Performing a query, the user can find all claims which are not supported. If the network was constructed in such a way that the bulk of the argument is made out of claims, then such a query would be useful. By looking at all unsupported claims, the author can decide which ones need further support and which others are strong enough to stand by themselves.

In order to find other types of objects for further elaboration, the user can simply query or look for nodes which are connected **to** other nodes, but not **from** others. These nodes, called *leaf* nodes, are the text objects which need to be self–explanatory. If there is a leaf in the network which can not stand on its own merit, then it most likely requires further explanation or support.

Unfortunately, one can not determine the entire meaning of a Euclid argument simply by looking at the nodes and connections. Throughout the analysis process, the user will need to actually read the contents of the nodes to understand the gist of the argument. With this in mind, one can realize that the only way to fully support an argument is to have substantial claims. If a claim is supported by other claims which are weak, then the author must continue to strengthen the supporters or add more resolute claims to the argument.

Overall Structuring

When an author writes an argument using Euclid, she should separate ideas and have clear connections between them. The system can not enforce this rule, but arguments which are written with this convention are more clear to their readers. When adding statements to an argument, one can fall into a few traps.

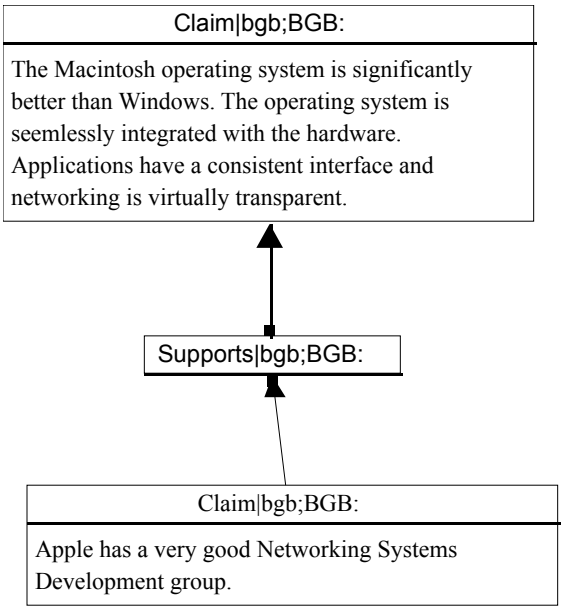


Figure 19. An example of a small argument which does not individualize its claims and is accordingly not clear. The supporting claim in this example supports only part of the main claim.

The author may wish to support a statement that contains a conjunction of ideas: perhaps it is several sentences with

different claims. Any relation connected to this statement would be ambiguous since a relation can only connect to a complete object.

Supporting only a part of the object would make the structure difficult to follow from the reader's perspective. Figure 19 illustrates an ambiguous connection between two claims.

Conversely, the author might make partial statements in order to have small independent sub–claims for each piece of an issue. If the granularity of the objects is too small, the reader would have the arduous task of sorting out enough claims to make a single comprehensible sentence. Both of these cases are burdensome to the reader; and remember, the author is usually the first reader.

When the author is reading through his argument, the logic should be straightforward. Consequently, lack of logic is equally clear. Euclid encourages writers to clarify and justify claims by making lack of logic obvious, especially to the author. When the author notices that something is lacking, he can add more claims to strengthen the argument.

In conventional writing, several ideas can be contained in the same paragraph or section, but the author may not realize that the statements which are supporting one part of a paragraph are not related to the rest of it. It is likely that the author will abandon some important claims without fully supporting them. A claim which may be very valuable to an argument can become orphaned and consequently vulnerable to critical readers. When writing in Euclid, it is very difficult to miss orphaned claims. If an author creates a claim which is not directly related to claims nearby, then it will stand out as the only claim without connections.

An author needs to be careful to have cogent arguments at all levels so that the reader can be persuaded that all parts of the argument are conclusive. It is easy for a person to write an argument with the assumption that the audience agrees with some obvious claims. Using Euclid, each claim stands out independently from the rest of the argument, so the author may take more time to be sure if claims may or may not be taken for granted. The author can read the claim, perhaps as an unsupported entity, and determine if it can remain an independent idea or if it needs further support.

4.2.3. Linearizing Structure

Once the logical structure of an argument has been constructed using Euclid, the linear form may be created by forming a reasonable order to the objects. List objects may contain the linearization of the argument. Objects may be added to a list object by moving them into it, and then the order of the list's members may be changed.

When the user is ready to export the complete argument, the list containing the whole argument or a linearized section of the argument may be copied as a whole, and pasted into a word processor.

Euclid supports copy and paste in several forms. Copying a group of objects from a display may result in:

- paste of the same objects and internal lines into another display.
- paste of the picture of the structure into another application.
- paste of the text of the objects into another application.

Using this copy/paste interface between Euclid and a word processor, an author can transform the logical, network structure of the argument into the sequential, written form.

5. Future Directions

We have been using Euclid extensively in our everyday work with some excellent results. Having gained some expertise in the use of the program, we have discovered some shortcomings which can be solved with some additions to the system.

If list objects had the ability to store layout information, then they could be used as iconic representations of sections of displays. In the current state, when a segment of an argument is stored in a list, the expansion of the list is poorly layed out, and is not a structure that the user created.

The program implements styleable text for the object contents. A useful feature would be to allow pictures, or other data types to be stored in the objects as well. Since Euclid supports the idea of graphical representation to help visualize ideas, it would be fitting for the program to support additional graphical entities.

While creating large displays, we have found that the screen eventually becomes a tangled *spider web* of lines and nodes. One reason for this is that for every relation, there are at least two lines; one entering the relation, and another exiting it. If we had a method for making direct links between objects, it might reduce the clutter. A representation of relations where the line color or pattern represents the type of connection could be a feasible solution.

A future version of the system needs to have network support. When a standardized store–and–forward messaging protocol becomes available on the Macintosh, this application would be a very good candidate to take advantage of it.

6. Conclusions

This paper was written with the help of the Euclid system. The author wrote the initial argument by brainstorming ideas and incrementally developing the structure. His advisor was able to understand the logical connections clearly and made meta–comments relating to the exact locations of the problems in the argument. The advisor added comments and corrections while the student continued to work on the same database. After the author merged the comments back into the his database, he was able to discuss the argument at the meta level while expanding the primary argument. This collaboration was very successful.

We are convinced from our own experience that Euclid is a powerful system, but now we need to determine how useful it is to the general public. We are currently developing tests to study the actual effectiveness of the system. Several experiments will be performed to investigate the success of Euclid for different applications.

The system exceeded our expectations in several ways. It enhanced our ability to construct clear, concise arguments, and it improved our collaborative effort by serving as a platform on which to discuss the argument effectively. We hope to convince our audience that Euclid can be a beneficial tool for all who read or write reasoned discourse.

Acknowledgments

I would like to thank Paul Smolensky, my advisor for all the time and effort he has put into the development of the program and this paper. The original concepts of the system were developed by Paul Smolensky, Barbara Fox, Roger King, and Clayton Lewis in 1986. Other contributors to the evolution of the project include Charles Hair, Brigham Bell, and Elizabeth O'Dowd. Additional thanks are extended to: Apple Computer, Inc. who helped give me *the power to be my best* (funding and equipment); The National Science Foundation, for grant IST–8609599; and Symbolics, Inc.

References

Apple Computer, I. (1987). *Hypercard User's Guide*. Cupertino, CA: Addison–Wesley.

Bernstein, B., Smolensky, P., & Bell, B. (1989). Design of a Constraint–Based Hypertext System to Augment Human Reasoning. In *4th Annual Rocky Mountain Conference on Artificial Intelligence*, (pp. 21-30). Denver, CO.

Conklin, J., & Begeman, M. L. (1988). gIBIS: A Hypertext Tool for Exploratory Policy Discussions. *ACM Trans. Office Information Systems*, 6(4), 303-331.

Halasz, F. G. (1988). Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems. *Communications of the ACM*, 31(7), 836-852.

Jarczyk, A. P. J., Löffler, P., & Shipman, F. M., III (1992). Design Rationale for Software Engineering: A Survey. In *Hawaii International Conference on System Sciences*, Hawaii.

Marshall, C. C., Halasz, F. G., Rogers, R. A., & Jannsen, W. C., Jr. (1991). Aquanet: a Hypertext Tool to Hold your Knowledge in Place. In *Hypertext '91*, (pp. 261-275).

Nielsen, J. (1990). *Hypertext and Hypermedia*. San Diego, CA: Academic Press, Inc.

Perelman, C., & Olbrecht–Tyteca, L. (1971). *The New Rhetoric*. Notre Dame, Indiana: University of Notre Dame Press.

Rieke, R. D., & Sillars, M. O. (1984). *Argumentation and the Decision Making Process* (2 ed.). Glenview, IL: Scott, Foresman and Company.

Schrage, M. (1990). *Shared Minds: the New Technologies of Collaboration*. New York: Random House.

Smolensky, P., Bell, B., Fox, B., King, R., & Lewis, C. (1987a). Constraint–Based Hypertext for Argumentation. In *Hypertext '87*.

Smolensky, P., Fox, B., King, R., & Lewis, C. (1987b). Computer–Aided Reasoned Discourse, or How to Argue with a Computer. In R. Guindon (Eds.), *Cognitive Science and its Implications for Human–Computer Interaction* Hillsdale, NJ: Lawrence Erlbaum.

Toulmin, S. (Ed.). (1958). *The Uses of Argument*. Cambridge, UK: Cambridge University Press.

Walton, D. N. (1989). *Informal Logic: A Handbook for Critical Argumentation*. Cambridge: Cambridge University Press.