Defining Software Visualizations Dynamically

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912-1910
(401)-863-7641, FAX: (401)-863-7657
spr@cs.brown.edu

ABSTRACT

In this paper we describe a system that allows the user to rapidly construct software visualizations over a variety of data sources for software understanding. The system provides the user with a visual front end that supports the definition of queries over multiple data sources without knowledge of the structure or contents of the sources and with a variety of back end visualizations. It produces a high-quality, easy-to-define software visualization that can address specific problems quickly and efficiently.

KEYWORDS

software visualization

1 BACKGROUND AND MOTIVATION

Software Visualization is the process of providing visual representations of a program and its execution to the programmer. Because software developers typically draw diagrams to describe and help others understand how their software works, the classical motivation for software visualization has been that it is an aid to software understanding. Software understanding is the task of helping a programmer to answer questions about the software during design, maintenance or development. It is a key to software development since it involves the ability to answer the specific questions that tend to arise in these phases. For example, a developer might want to know why a particular function is called so often or how a particular situation involving timing constraints could arise or what needs to be modified to add a parameter to a given function. For a variety of reasons, software visualization has not fulfilled its promise for software understanding. In this paper we provide a brief analysis of these reasons and describe a new visualization system, Cacti, that is designed to be used for software understanding.

1.1 Software Visualization Results

Software visualization has been quite beneficial as an aid to navigation. Our studies [14] and reports from FIELD users demonstrated that this use was by far the most important application of the various diagrams that FIELD provided. Moreover, this lesson has been taken to heart by environments such as Microsoft's Visual C++ where the only visualizations that are provided are small hierarchical displays of file contents or the class graph shown almost exclusively for navigation purposes.

Specific software visualizations have also been quite successful in addressing some explicit problems. The FIELD memory visualization has been used effectively for finding memory leaks and understanding anomalous program behavior. The various performance visualizations provide a more compact and easier to understand display of performance information and are widely used for this purpose. This is especially true in the more complex domain or parallel performance where textual displays and listing have problems conveying the time-dependence of the relevant information.

Software visualization, on the other hand, has not been widely used for understanding. The results of our study showed little if any effect on software understanding from using visual tools. Similar experiences and the fact that these tools, while practical, are not widely used and have not been widely adopted, show the same thing. Our concern in renewing our efforts in software visualization was to understand why previous efforts have failed and to see whether the roadblocks to such success could be addressed.

Our analysis has identified three reasons why software visualizations have not succeeded for software understanding. The first problem is that today's visualizations do not address the specific questions that are inherent to software understanding, i.e. they can not display the information that is relevant to the question at hand and discard the remaining information. The second problem is that current visualization techniques are not capable of displaying the large amount of information or complex mutli-dimensional relationships that are needed for software understanding. The third problem is that today's software visualizations are too difficult to set up, requiring extra work on the part of the programmer to generate data for any visualization and demanding a significant amount of programming to implement a custom visualization.

1.2 Overview of our Approach

To create a practical approach to software visualization we need to deal with the three problems cited above. The Cacti system we are developing, and the Desert environment [23] it is a part of, attempt to accomplish this by 1) providing a range of automatic and inexpensive data collection techniques, 2) offering a range of back-end visualization methods, and 3) providing the facilities to allow the user to define high-quality visualizations quickly and easily. While the first two of these techniques are covered elsewhere, the third is the primary topic of this paper.

The first component of our solution is to gather the information to visualize using automatic and inexpensive data collection techniques. We achieve this through a variety of techniques, most of which have been implemented in our previous environment, FIELD, in our current programming environment, Desert, or in an experimental project, AARD.

Most compilers today will generate cross reference information. For example, Visual C++ generates *.bsc files while Sun's compilers generate source browser files. Our approach is to piggyback on the compilers by using these files with our own conversion tools running in background to translate the resultant data into a common format and extract more detailed semantic information. This information is augmented by fast scanners included in Desert to extract basic information not available from the compiler output. These scanners identify logical units of the source files, identify the top level components in files that are not ready to compile (i.e. while files are being edited or created), and handle other software artifacts such as object design diagrams or user interface definition files.

Access to this information is provided by a set of background database processes that automatically update as the underlying files change [25]. These processes provide shared databases for common projects and local databases for single-user projects. They offer an extended SQL-based interface to applications such as a visualization engine. The underlying databases are viewed as relational.

The FIELD environment provided additional data stores for program information. The *formserver* back end used various versions of the UNIX *make* and *rcs* tools to determine file dependencies and provided a queryable interface to this information. The *profserver* back end used different UNIX performance evaluation tools to gather performance information about a binary and offered a simple interface to return this information to other tools. The *symtblserver* back end reads the symbol table from an executable file and allows queries of the relevant information. All these run when necessary without user intervention.

We have developed other data collection tools as part of the AARD project. These tools are based on dynamic modifications to a running binary to gather run time information similar to that done by MIPS' Pixie, Purify [8], and others [10,11]. These tools have been used to implement an instruction count profiling package, fast call tracing, heap utilization tracing, and full memory tracing. We are currently in the process of organizing these tools so that they will provide a queryable interface to the gathered data that can be used for visualization.

A practical visualization facility must provide a wide range of visualization strategies. This allows an appropriate strategy to be used for each specific visualization to maximize the amount of information presented and to highlight the important relationships. Our previous experience with software visualization indicated that, especially for large visualizations, abstract representations such as those provided by Seesoft or the performance visualizations of AARD provide more information in a limited space than do graphs. The experiences of the Xerox information visualizer and our own experiments with 3D visualization have demonstrated that using the third dimension can increase the amount of information that can be presented effectively by about a factor of ten. Because the field of interactive information visualization is quite young, new visualization techniques are continually being developed. It is important that any practical software visualization system offer an extensible variety of two and three-dimensional visualizations strategies.

At the same time, the visualization back end must provide powerful techniques for browsing over the resultant visualization. The purpose of the overall visualization is to provide an overview of the data. The user needs to use this overview to understand relationships and then to concentrate on particular items of interest. Browsing here should be done using implicit hierarchies, user selections, naming conventions, and any other techniques that are available. Providing high-quality browsing facilities also eases the burden on the user when defining the visualization since it will be easier to specify an approximation to the desired information and then to use the browsing capabilities to narrow the focus once the visualization is available.

Our previous visualization work, with Garden [19], FIELD, and our more recent three-dimensional efforts, Valley [22], provide a framework that accomplishes both of these tasks. The more recent work provides an extensible framework that allows the easy incorporation of a wide variety of visualization strategies. We have currently integrated about ten different strategies and are able to add a new strategy with a day or two's effort. Moreover, these facilities provide a variety of browsing techniques that utilize multiple hierarchies, naming, user selection, and implicit and explicit relationships. We are using an updated version of Valley as the back end for the Cacti visualization system described here.

Once we have a variety of information available for visualization and a broad range of visualization strategies that can be used for realizing visualizations, the key to producing a practical software visualization environment is to provide a powerful front end that allows the user to quickly define the appropriate visualization for the task at hand. To be successful this front end must:

- *Provide access to all data sources*. The real power of visualization comes to play when multiple data sources, e.g. performance data and the underlying program structure, can be integrated in a seamless manner. The front end should allow the user to select from and integrate data from the different sources in a unified manner.
- *Allow simple selection of the data to visualize.* The selection of data should not require the user to understand the format or the source of the data. In particular, the user should not be aware if the data is stored as a straight file or in a database, and, if it is a database, the user should not need to know the schema of that database.
- *Utilize a straightforward query language*. While the problem of selecting the data is essentially that of defining a query, the user should not be forced to construct queries in a non-intuitive language. Rather the query interface should be as simple and intuitive (and visual) as possible.
- *Allow easy selection of the visualization strategy.* Once the data to be visualized is defined, it should be simple for the user to choose between the different visualization strategies and construct an appropriate visualization of that data.

The rest of this paper describes the Cacti software visualization system. In developing Cacti we attempted to meet the above criteria by developing a front and back end that offered the user a variety of data sources, a very simple query interface, and flexible visualization strategies. We begin in the next section by describing our visualization model based on objects, relationships, and parameterized visualization strategies. This is followed by detailed discussions of how the user can define objects using the Cacti interface and how the visualization is selected. We conclude with a brief discussion of the implementation and our experiences with it.

2 OUR APPROACH

2.1 An Example

Digital Equipment Corporation had a problem with understanding C++ programs. Inefficiency and potential bugs are introduced in such programs when the compiler creates class temporaries for call parameters or within an expression. They wanted a tool that could find all instances of such temporaries over a large system and allow the user to browse over the result to find code fragments that should be changed.

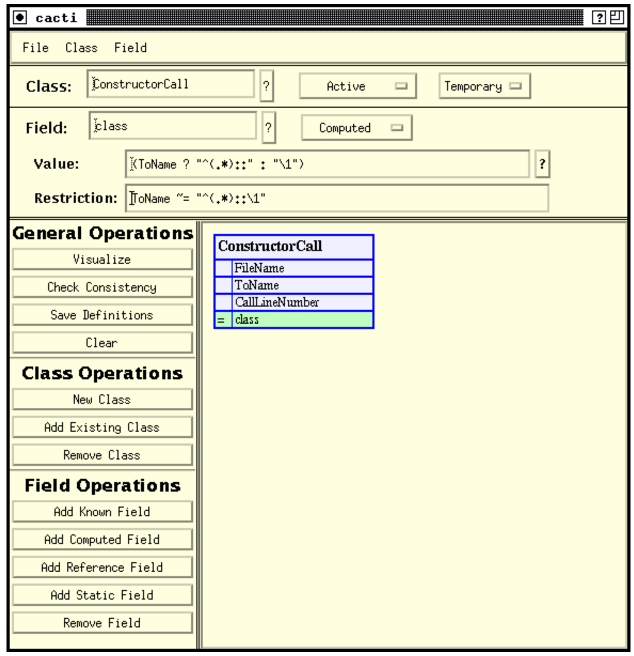


FIGURE 1. A view of cacti showing the definition of a class of objects corresponding to constructor calls within a system.

Rather than creating a separate tool, we can define an appropriate visualization using Cacti. A temporary can be identified by the existence of a constructor call and a destructor call for the same class at the same line in some source file. We use Cacti to create two classes, one to represent constructor calls and one to represent destructor calls. Figure 1 shows the first step in this process, with the `ConstructorCall` class of objects defined. This was built by first selecting new class and then choosing the file name, to name, and line number from the set of known fields displayed in a dialog window, then defining the `class` field as a computed field based on the `ToName`, and restricting the `ToName` field to be a constructor name. The set of known fields is the union of data fields from all available data sources. Note that the user does not need to know the structure of this data or its form in order to use the system.

The second step in creating the visualization is to create a similar class for destructors, repeating the above operations but this time restricting the `ToName` field to be a destructor name. After this is done, the two classes are related using a reference field as shown in Figure 2. Reference fields provide a way of relating information in one class to that of another. In this case, we specify that a constructor must be found with a matching file name, line number, and class name or the corresponding destructor call entry should be deleted. Here we have also designated the `ConstructorCall` class as passive so that only the remaining destructor calls, those that reflect temporaries, will be display.

The next step is to request that this data be visualized. The user does this by clicking on the Visualize button. At

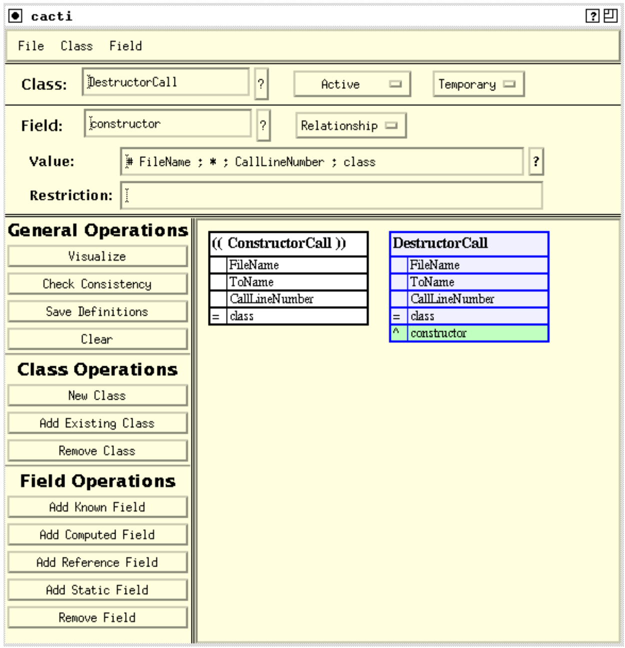


FIGURE 2. The complete description of the visualization of class temporaries.

this point, Cacti uses a variety of built-in mappings and heuristics to determine how to map the class specification given by the user into a set of queries on the underlying databases. If the result cannot be determined or if it is ambiguous, the user is asked for clarifications. Otherwise, Cacti looks through the set of available visualizations and determines which of these might be appropriate for the given set of objects. If more than one visualization is appropriate, Cacti asks the user to select the desired visualization style. In this case, we choose a `FileMap` style that provides a mapping of information to source files and works over a large domain.

At this point, Cacti puts up the dialog box shown in Figure 3 to allow the user to specify parameters for the visualization and the mapping from fields to visualization information. In this case we have specified that the backdrop color be white and that the remaining parameters be assigned default values and presented to the user as dynamically settable options as part of the visualization. Moreover, we have specified which fields of the `DestructorCall` class should be viewed as containing the file name, line number, and data statistic that is required by a `FileMap` visualization. In this case, we specified that the statistic should be the class name.

Once we accept these parameters, Cacti runs the visualization engine Mirage with a data file describing how to get the appropriate data and how to display the result. This data file can be saved to allow the same visualization to be applied to different systems or at different times without having to rebuild the description in Cacti. Mirage contacts the necessary databases, extracts and combines the data

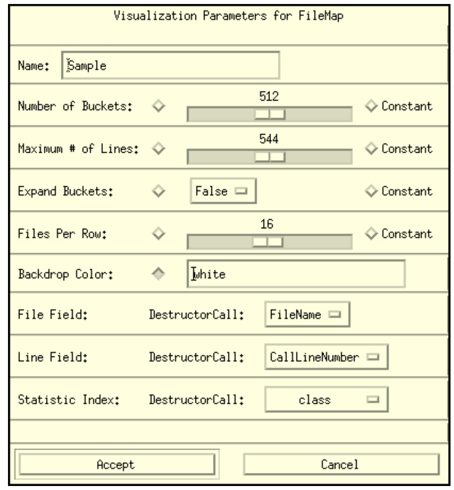


FIGURE 3. Cacti's dialog box for defining the visualization of class temporaries.

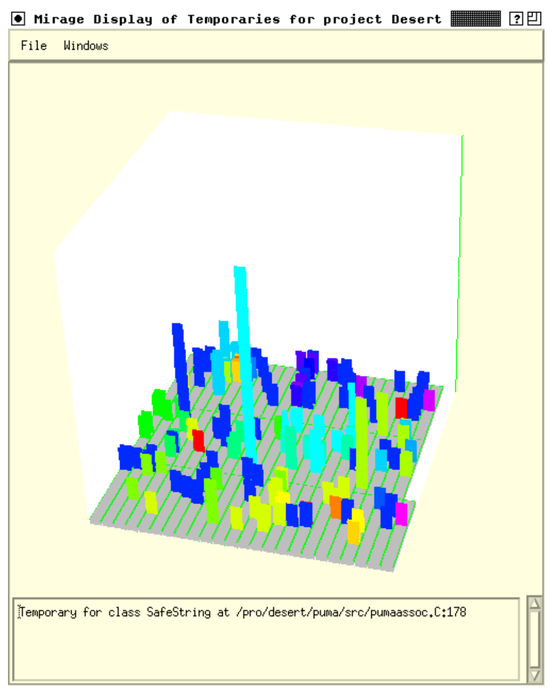


FIGURE 4. The resultant visualization of class temporaries in the Desert environment.

from possibly multiple sources, and then uses the underlying visualization engine to put up a display.

The display corresponding to this example can be seen in Figure 4. The visualization displays each file as a row split into buckets for the different lines. In this case, because there are a large number of files, multiple files are shown in each row. An instance of a temporary is shown as a box on top of the file displays. If multiple instances occur within the same bucket, these are stacked up on top of each other in such a way that a top-down view would show all the different colors within a bucket. The statistic field is used to determine color. Mirage automatically determines that the value specified in this case, the class name, is a string, and maps this into integer values for the visualization which then use the values to determine the color of the different nodes. Thus each class for which temporaries are created is given a different color. The relative preponderance of one color indicated a large number of temporaries of that class type. This is particularly useful in this case since temporaries of some classes are benign which those of others indicate real or potential problems. By simply looking at what colors correspond to what classes, the user can quickly identify where the actual problems might be.

2.2 The Visualization Model

While there are a variety of designs that could be used for general purpose software visualization, we have chosen a strategy that separates the definition of what should be visualized from how it should be visualized. This has allowed us to deal with visualization issues, i.e. different visualization strategies, browsing techniques, and graphical support, independently of the issues of what data should be visualized.

Our visualization back end assumes that the data to be visualized has been organized as sets of objects and relationships. Objects are viewed as collections of fields; relationships consist of connections that relate two objects. The visualization engine maps the objects and relationships into corresponding graphical objects, components and constraints based on a visualization specification that describes the type of visualization, the types of objects in that visualization, and the mapping from user objects to visualization objects. The relationships can be used to construct graphical objects (such as arcs) and are used to define hierarchies for browsing. In addition, each visualization strategy is parameterized. The parameter values can either be predefined or can be set dynamically by the user to affect the visual presentation. This is described in more detail in [22].

Given this model for the visualization back end, the function of the front end is two-fold: to generate the sets of objects and relationships and to choose the visualization strategy to be used along with its various parameter settings and the mapping from objects to visualizations.

Objects for visualization need to be built from a variety of data sources in a standard way. Relationships can be viewed as special types of objects, or more simply, as fields in those objects that are essentially pointers to other objects. Given this view, the problem of constructing the appropriate set of objects is essentially a database query problem where the data sources vary from actual databases to dynamically generated trace data. The important point, from our perspective, was that while users would be defining such queries, they should not be aware of the underlying data structures, should not be forced to use a query language, and should be provided with a simple and direct interface that allows the task to be done quickly and accurately. How we achieve this is described in the next section.

Once the user has defined a set of objects, the visualization system needs to define the visualization specification. We use a resource file to describe all the available visualization strategies. Each strategy description contains the parameters of the visualization and information for mapping data objects to their visual counterparts. The latter is organized so that the front end system can readily determine which visualization techniques might be appropriate to the user's defined objects. The user is then asked to select the technique to be used. Once the technique is selected, the user is asked to define initial values for parameters and whether those parameters can be changed during the visualization. The actual mapping from data to visual objects is mapped into a set of parameter settings for this purpose. This is described in the following section.

3 DEFINING THE QUERY

In order to simplify the definition of objects by the user, we needed to develop a unified data model to describe the variety of data sources, a common model for representing the target objects, and an algorithm that allows us to define a query building the target objects from the underlying data sources. To simplify the user's task in defining objects, we decided to use a variation of a universal relation model with a visual front end. While there are a number of visual query languages [4,6,30], those that have been proposed required too much knowledge of the underlying databases for our purposes. By hiding the structure of the underlying data, we intend to greatly simplify the user's query definition. However, doing so requires a more elaborate underlying data model and a heuristic algorithm for actually defining the query from the user's specification.

3.1 A Unified Data Model

For simplicity and consistency, we view each data source as a relational data store. That is, each database consists of a set of relations with each relation consisting of a set of tuples. Each tuple contains data for one or more base fields, with all tuples for a given relation containing the same set of base fields. The base fields in this case contain raw data (i.e. numbers or strings) and not pointers to other objects. This approach seems to be sufficient for the data sources that we currently have available. Our sources of cross-reference data are currently organized as relational data. Build dependency and profiling information can easily be viewed as a relational database consisting of a single relation containing the information of interest. Similarly, trace data can be viewed as a single relation where each trace item contains appropriate descriptive fields and an event counter. None of our current data sources contain pointers (such as an object-oriented database). If there were one, we would incorporate it into this scheme by replacing the pointer with one or more data fields that uniquely identify the object pointed to.

The key to our representation is that we associated each base field with a domain. Domains provide a consistent internal type structure that allows the system to relate common data across multiple relations and across multiple databases. Domains are arranged in a hierarchy that represents generalizations. For example, the name field of the `Ref` relation is of domain `RefIdName` which is a subdomain of `IdName`. Similarly, the name field of the `Def` relation is of domain `DefIdName` which is also a subtype of `IdName`. This allows the user to choose `IdName` where either the definition or the reference may be appropriate or `RefIdName` if the reference name is to be used explicitly. It also allows the underlying system relate the `Ref` and `Def` relations using the corresponding names.

The domain identifiers represent the interface provided to the user. Instead of selecting base fields from the various relations, the user selects the appropriate domains. The system then maps the domains into the most logical base field based on the overall query using a set of heuristics described later in this section. The fact that two base fields share a common domain is used as an indication that they can be used to relate two different relations. Additional information on how the various relations are associated can be explicitly provided to the system as mappings. For example, the relationship between a reference and its definition can be specified so that a tuple in the reference relation is mapped by default to its corresponding definition rather than to any definition with the same name, type and file. The underlying system relies on these explicit mappings to insure a consistent interpretation of the underlying data sources.

All this information is provided to Cacti in a set of resource files that describe the underlying data sources. This provides a flexible interface that allows new data sources to be easily incorporated into the system and allows the system to be adapted to other applications. An extract from the cross-reference database specification is shown in Figure 5.

```
DATABASE SXRFDB:
    ACCESS "SAND(SXRFDB)"

RELATIN Def ::
     scope : String => DefScopeName < ScopeName
     name : String => DefIdName < IdName
     type : Integer => DefIdType < IdType
    scope_type : Integer => DefScopeType < ScopeType
    line : Integer => DefLineNumber < LineNumber
    file : Integer => DefFileId < FileId

RELATION Hierarchy ::
    from : String => SubClassName < ClassName < IdName
    to : String => SuperClassName < ClassName
    line : Integer => HierLineNumber < LineNumber
    file : Integer => HFileId < FileId
    virtual : Boolean => HVirtualFlag
```

FIGURE 5. Extract from the data source resource file for the cross reference database SXRFDB. The Access line indicates how data from this database should be accessed. Two relations are defined here. Each consists of base fields. For each base field the file indicates the name, its internal type, the domain of that field, and superdomains of the domain. Additional information that can be specified includes mappings between relations and semantic information relating the fields within a relation.

3.2 The User Object Model

While the underlying data model is relational, the model presented to the user is object-oriented, consisting of a set of class definitions defining the sets of objects to visualize. Each class consists of a set of data members. There are four types of data members:

- *Domain members*. A domain member corresponds to a single domain from the underlying data model. These represent base data that can be derived from any of the base fields that correspond to that domain or any of its subdomains. For example, just from the specifications of Figure 5, a domain member referring to *IdName* could represent `Def.name`, `Hierarchy.from` or `Hierarchy.to`.
- *Computed Members*. These data members are defined using expressions that can range over constants and other data members of the same class. The restriction to data members of the same class insures that the specification of a class remains consistent. Additional domain members can be added to a class where necessary just to serve as data for a relevant computed member.
- *Static Members*. These represent dynamically settable constants. They are not actually a part of any resultant class. Static members can be used as constants in a computed member or in a restriction (defined below). They are assigned an initial value by the user when the visualization is defined. However, when the visualization is presented, the user has the ability to reset this value and thereby cause the visualization to be recomputed. Static members can be used to set thresholds on what should be displayed and to identify focal points for the visualization.
- *Reference Members*. These are used to define links between classes and hence to define relationships. Each reference member refers to an object in another class. In defining the relationship, the user must specify a match value or a don't care for each field of the class that is referred to. These values, for simplicity purposes, must be other members of the class where the reference member is defined and not arbitrary expressions. (Note that expressions could be used by simply defining a computed member.) Reference members can either be pas-

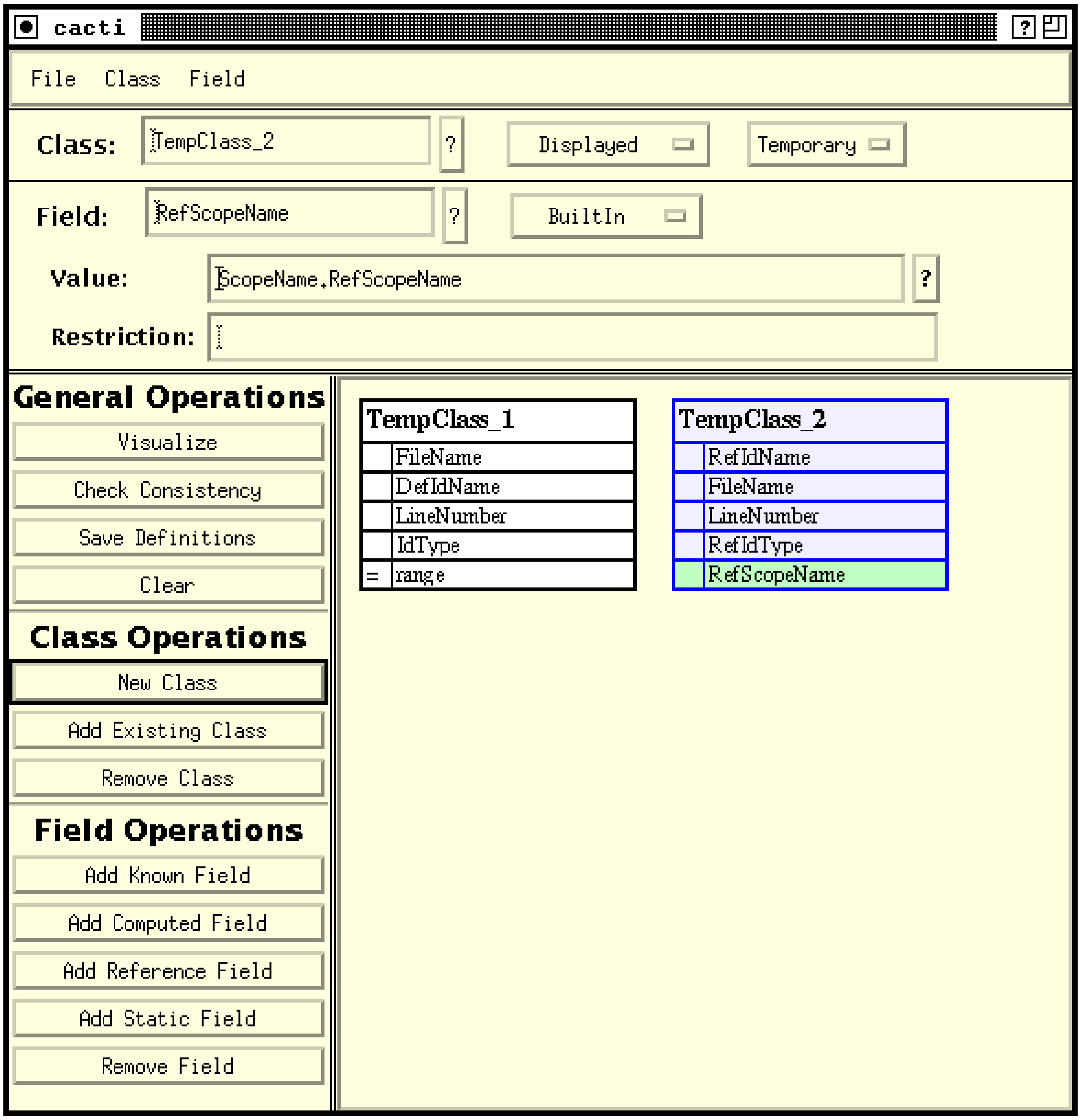


FIGURE 6. The Cacti user interface showing a user model. The window consists of a menu bar, panels for displaying and editing properties of the current class and member (called a field in the interface), button panels for common operations, and a visual display of the current user object model.

sive or generative. A generative member will cause a new object to be added to the referent class if no match can be found. Here, if a match value is a don't care, a default value must also be specified for building the new object.

In addition to the definition of members, each class can have a set of associated restrictions. In the front end these are associated with individual members, but internally, all restrictions are viewed as relating to the class as a whole. Each restriction is a Boolean expression that must evaluate to true for a valid object of the given class. Restriction expressions again are constrained to constants (and hence static members) and members of their own class.

The Cacti user interface corresponding to this model can be seen in Figure 6. The overall window is divided into seven regions. The top pane is a menu bar containing most commands on pull-down menus. Immediately below this is a class panel. Here the user can view and edit the properties of the currently selected class. This class can be selected either by clicking on it or by choosing its name in the class panel. The two buttons in the class panel allow the class to be part of the definition or not, to make the class sequential and hence animated, and to make the class part of any future saved definitions.

Below the class panel is a field panel for displaying and editing properties of the currently selected member. (Members in the model are referred to as fields in the user interface.) Fields can again be selected either by clicking on them or by choosing their name in this panel. The value button is specialized to the different types of members. Currently the user must type in expressions and restrictions, but we expect to provide more convenient dialog boxes in the future.

Below the field panel on the left are three button windows providing fast access to common operators. The Visualize button under general operators constructs a visualization from the current model; the Check Consistency button checks whether the current model has a valid associated query; the Clear button clears the model. The class operators allow the user to manipulate the set of classes that are part of the model. Similarly, the field operators allow the user to manipulate the set of fields in the current class as well as static fields.

The core of the interface, beyond the panels and buttons, is the visual display of the current user object model in the remainder of the window. Here each class is displayed with all of its members. The box to the left of the field is used to indicate the member type (empty is a domain member, = indicates a computed member, ^ indicates a reference member). Static members are shown as part of a special class. Class types are indicated by shading in the class name region. The user can select members of classes by clicking on them with the mouse. The currently selected class is shown with a light-blue background, while the currently selected member has a light green background.

3.3 Building a Query from the User Model

The user model described above provides a convenient mechanism whereby the user can define the data to be visualized. It satisfies many of our objectives, i.e. it does not require the user to be aware of the underlying data stores, it does not introduce a complex query language, and it is primarily a visual representation. The problem that arises with this formulation is that the representation does not define a query in an unambiguous manner. There are two sources of ambiguity. First, each domain member can correspond to a variety of different base fields. Second, if different base relations are referenced by members of a class, there may be multiple ways of joining these relations.

Our approach to this ambiguity is to attempt to select the query that is most "natural" and therefore most likely represents what the user intended. To do this we associate weights with different ways of relating two underlying relations, and then attempt to find a minimal cost solution that assigns each domain member to a base field and that joins all the referenced relations. This problem is inherently NP-complete [9], but our cases are relatively small and a branch-and-bound approach is feasible. Moreover, it can be the case that two solutions are equally "good" (or close thereto), and we want to ask the user which was intended. Thus we actually need to find all solutions within a given bound of the optimal one. The cost model we are currently using is shown in Figure 7.

Cost	Situation
1	Domain member whose base field is in a relation that is already used.
5	Computed member.
100	Domain member whose base field is in a relation that hasn't been used.
50	Two relations associated via a mapping specified in the data model resource file
50	Two relations associated using domain relationships
10	Two relations associated using a user restriction
10	Delta for which two selections are considered identical

FIGURE 7. The current cost values used in determining the appropriate query for a given user model. The first three costs are added to the total cost when a member is used (static and reference members are ignored). The latter three costs are used when relations used by the base fields need to be related to each other.

While there has been significant theoretical criticism of the universal relation assumption for general databases, we feel that our use of domains augmented with the ability to define explicit mappings between relations and the fact that we are working in a well-understood domain where we have control over the underlying databases, allows us to circumvent these problems and create a practical interface.

The algorithm we use resolves one class at a time. The classes are first ordered topologically based on reference members so that when the objects of a class are created, it will be possible to define the reference fields as well. Each class is then considered independently.

The first step in resolving a class is to try out each possible assignment of base fields to domain members. This is done one field at a time. First, the field that has not be resolved that has the minimum expected cost is chosen. If this is a domain field, then the possible base fields are ordered so that those with the minimum expected cost will be considered first. Then each possible base field is considered in turn and all potential solutions with that field are considered. At all times, the algorithm keeps track of the best solution that has been found. If the actual cost of a new solution being constructed is greater than the cost of the best solution plus a delta, then the new solution is discarded.

The task of building an actual solution for a given assignment of domain members to base fields is more complex. The basic problem here is to take the set of relations that are used by the selected base fields and determine how these might be related to each other as part of the query. One possibility is that there might be a user restriction given for the class that associates items in one relation with items in another. This is a preferred choice as reflected in the cost model since it indicates the user's understanding of what the relationship should be. Alternatively, relations can be associated with each other either through explicit mappings defined as part of the data model or through fields that share a common domain. The algorithm does not have to find pairwise relationships between all the relations, but rather has to find a set of relationships so that each relation can be related to any other relation using a series of relationships.

Our approach views the possible pairwise relationships as arcs over the graph of the available relations. We first build such a graph internally for the particular query. Most of the graph is fixed; however the need to incorporate user restrictions means that the graph is actually specific to the particular class being resolved and to the particular assignment of domains to base fields. Then we compute all-pairs shortest paths on this graph to determine the minimum cost way of relating any two relations. The result is an augmented graph over the set of all relations. Next we compute the minimum cost tree that covers the set of relations referenced by the selected base fields using this augmented graph using a modified minimum spanning tree algorithm. This tree represents a set of relationships that both has a minimum cost and satisfies the necessary criteria for building a query. If the new solution is less expensive (relative to the specified delta) than the current best solution, then it is added to the set of available solutions and other solutions are discarded if appropriate.

When this algorithm completes, it yields a set of solutions each of which specifies an ordering for the members and an ordered set of relationships among the reference base relations. If more than one solution is found, the different alternatives are presented to the user and the user is asked to choose the appropriate query.

3.4 Generating the Objects

Once a class is resolved, its objects can be retrieved. The basic idea is to construct a database query using the solution. Before optimization, this query is formed algebraically by taking the relational product of all the base relations used, defining all the computed fields, selecting based on the domain associations and user mappings, selecting based on all the user restrictions, projecting onto the target members, and then outputting the tuples that result. The query as it is actually implemented must be optimized (e.g. using joins wherever possible), and must take into account the fact that the data can come from multiple data stores.

Our implementation uses a query engine that we built previously for data management in Desert. It offers an extensible set of relational operators, it provides both an algebraic and a SQL-based front end, and it will deal either with data in memory or on disk. It also includes a powerful optimization framework that can transform a raw query as described above into a form that can be efficiently evaluated. To use this package, we augmented the base package with a new type of relation that was a mirror of a remote data store. The task of generating the objects is then relegated to building a query as described above using the algebraic front end of this augmented query engine.

While we could build the whole query this way, it would not take best advantage of the capabilities of the remote data stores, some of which are efficient databases in their own right. What we actually do is to build a query for each external data store that is referenced in the solution. What can be included in this query is determined by the access description that is provided by the data store in the descriptive resource file. For example, the access description shown in Figure 5 for the database SXRFDB is `SAND(SXRFDB)`. This indicates that the database is a Sand database (Sand is the database component of Desert) and tells the back end that it can handle algebraic query specifications directly. Other access methods only allow all raw data to be obtained or provide a simplified indexed access method only.

The overall query is then built by making multiple passes over the solution structure. First there is a pass for each database. This pass gathers whatever information needs to be passed to the external data store and then asks that store to pass back the corresponding information in one or more temporary data files. The pass also constructs an appropriate database expression to access this data file and marks those parts of the solution that have been completed. After all these passes have been made, a final pass over the solution is used to build the actual database query that will be evaluated in memory to generate the objects. This is similar to the basic query idea described above except that it uses the access operators generated by the various database passes as the base and ignores any components that have been marked as completed. The result is a query that is efficiently evaluated within the visualization engine using the capabilities of the remote data stores as much as possible.

Once all the non-reference members of a class are defined, we are able to generate the reference members. This is done by making a pass over each generated object in the class

```
STYLE FileMap
REQUIRE
CLASS object MULTIPLE => FileMapItem(ExternalData)
 FIELD file : FILE
 FIELD line : NUMERIC
 FIELD data : NUMERIC

PARAMETERS
 NumBucket "Number of Buckets"  : INTEGER(1,1024) USER
 MaxLines "Maximum # of Lines" : INTEGER(64,1024)  USER
 ExpandBucket "Expand Buckets" : BOOLEAN USER
  FilesPerRow  "Files Per Row" : INTEGER(1,32) USER

 FileIndex "File Field" : FIELD FILE
 LineIndex "Line Field"  : FIELD NUMERIC
 StatIndex "Statistic Index" : FIELD INTEGER OPTIONAL

END FileMap
```

FIGURE 8. Visualization resource file definition of a visualization strategy. The first part describes the data requirements for this visualization style. The second part defines the visualization parameters.

and using the reference matching parameters to find the associated element of the referenced class. If a new instance has to be created, then one is added, and its reference members are also found. Note that we do not currently allow restrictions on reference members, but plan to do so in the future.

4 DEFINING THE VISUALIZATION

Once the data to visualize has been defined, Cacti needs to have the user define the appropriate visualization. The visualization framework we provide offers an extensible selection of visualization strategies, each of which is applicable in certain circumstances and each of which is parameterized. We wanted the system to provide the user with the set of relevant visualizations for the defined data.

To accomplish this and to allow the set of visualizations to remain extensible, we define the set of available visualizations in a resource file. There are two parts to the visualization definition in this file. The first describes the situations under which the visualization is applicable and the second describes the set of parameters that are associated with the visualization. An example visualization file definition is shown in Figure 8.

The first part of the resource file defines a data model for the visualization. The data model can identify one or more classes, each of which has an associated set of fields. For each class defined here there must be at least one data class defined by the user that matches. A class matches if it contains distinct fields that have the same types as the fields of the required class. A special set of field types was created for this purpose so that `FILE` corresponds to any internal type that indicates a file or file name and `NUMERIC` can be any numeric type. The keyword `MULTIPLE` allows multiple data classes to match and be considered for visualization. Finally, the requirement definition specifies the mapping from a user data object to a visualization object, in this case an `ExternalData` object inside a `FileMapItem` component.

The second part of the resource file description defines the parameters of the visualization. There are two types of parameters. The first type, represented by the first four parameters in the figure, describe the actual visualization parameters. These have a label and type and are used both when the user is specifying the visualization to define default values and, if the USER option is specified, when the visualization is being displayed to allow the viewer to adjust the visualization. The second type of parameter is used for mapping data members of the given classes to fields or values needed for visualization. In this case there are two required fields, one specifying the file name and one the line number, and one optional field specifying the statistic to be associated with that file/line pair. These parameters currently are not settable at run time. Moreover, unless the keyword OPTIONAL is specified, they must be defined by the user.

When the user has finished defining data and requests a visualization, Cacti first determines the set of visualizations whose requirements are met by the defined classes. If there is more than one, it asks the user which is appropriate. Then it creates a dialog box similar to that of Figure 9 allowing the user to set initial values for the parameters.

When the user clicks on accept the visualization appears. The actual visualization is generated by creating the appropriate root visualization object based on the visualization type. Then for each generated class, each object of the class is used to generate a visualization object and its associated component. We first do this generation for all classes that have not been marked as `SEQUENTIAL` and display the result. Then for each class that is sequential, we add the corresponding object to the display and redisplay the result. Because our back end visualization engine automatically animates between redisplays, the effect of this is to provide an animation where the tuples of the sequential class are added dynamically.

Figure 10 shows the visualization from the example of Figures 2 and 5. We are still working on developing more sophisticated mappings between the generated objects and visualization objects to take into account visualizations where a user object might yield more than one visualization object and where hierarchies of visualization objects need to be generated explicitly.

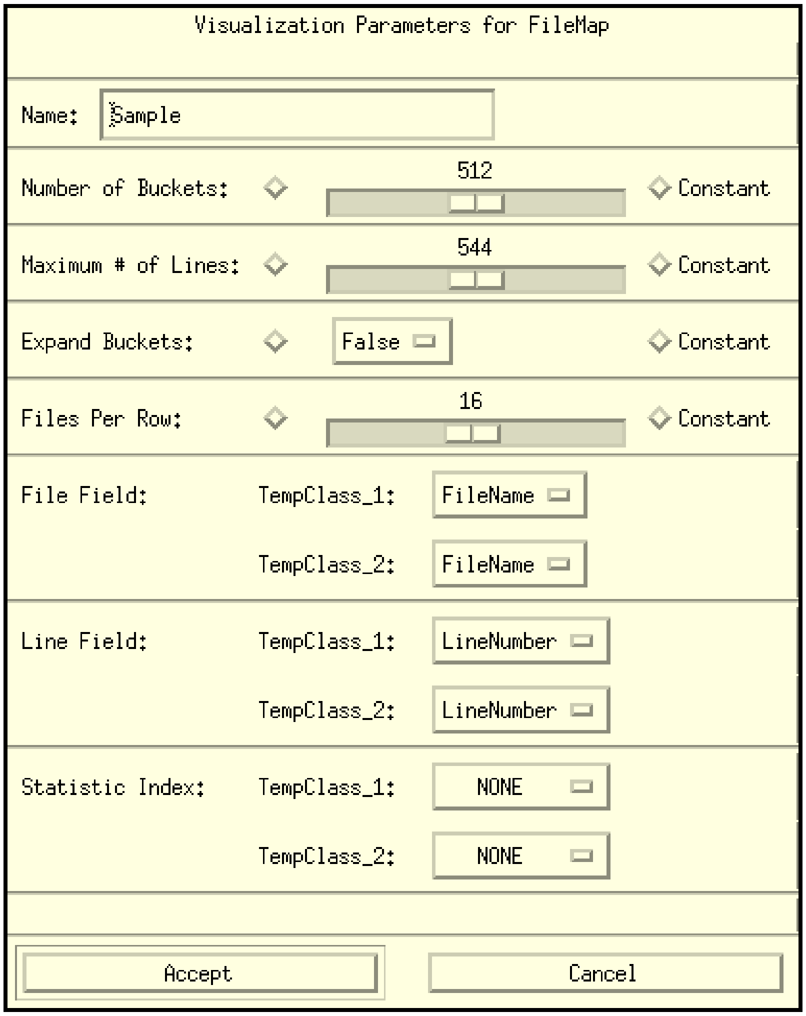
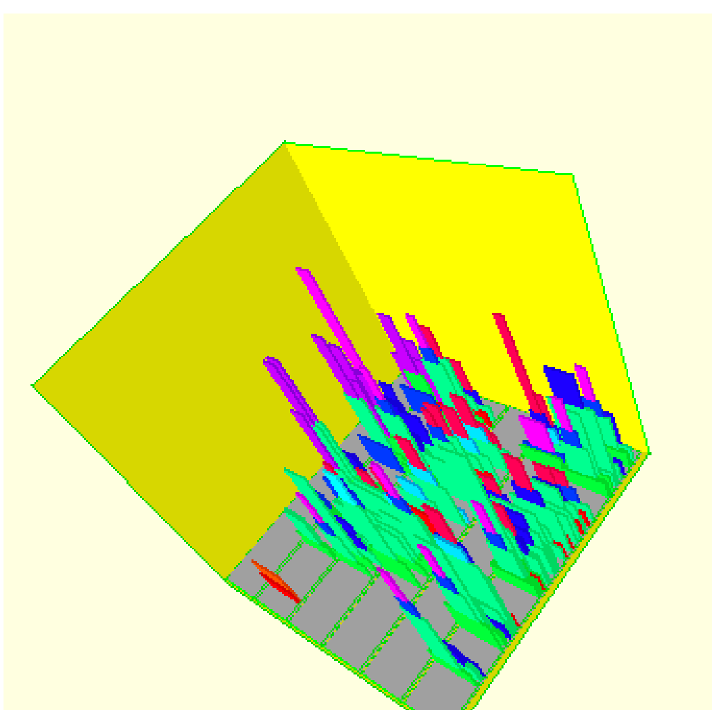


FIGURE 9. Dialog box for defining a visualization. The user can name the visualization, define initial values for the settable parameters, and associated members with visualization fields. Each non-field parameter allows the user to specify whether it is settable or constant at visualization time. Field parameters provide a list of fields of each class of the appropriate type. When the box is accepted the visualization appears.



4.1 Visualization Examples

5 RELATED WORK

Software visualization efforts have a long history, dating back to a variety of programs that would automatically produce flowcharts from a deck of Fortran cards [18,24]. More recent efforts have aimed at providing interactive visualizations of large software systems. The FIELD environment, for example, provided call graphs, dependency graphs, and a class browser, all of which were designed to handle moderately large software systems [21]. Similar tools have since been implemented as part of a number of commercial systems including ObjectWorks, DEC's Fuse, Sun's Workbench, HP's Softbench, and SGI's CodeVision. These tools provide views of the static program structure. They typically exploit the implicit hierarchies to provide browsing capabilities and use a variety of information encoding techniques to provide substantial information to the user. Other related work can be found in visual approaches to reengineering such as the Rigi system [16].

More dynamic visualizations can be seen in the display of data and related program structures. This work has taken two forms. The first involves data structure display, starting with the work of Brad Myers [17]. More recent work here ranges from FIELD where full data structures are displayed automatically using standard or user-customizable graphics and with automatic or semiautomatic update as the program executes, to the displays found in Sun's Workbench or ObjectWorks where only standard displays are available and the user decides what information to display and its screen location. The various efforts at algorithm animation represent a second approach to this problem [2,3,12,29]. Here the implementor crafts detailed displays that describe the underlying behavior of a specific program and its data.

Another category of software visualizations deal with understanding performance. Displays for sequential programs range from the typical histogram displays found in UNIX *prof*, FIELD, and most current software environments, to the sophisticated graph displays provided by Pure Software Corporation or the 3D displays we developed for the AARD project [20]. A large body of visualization research has been devoted to understanding the behavior of parallel or distributed programs [1,7,15,28]. Displays here have typically concentrated on one aspect of the problem, either processor utilization for or message passing.

Finally, there are a number of more recent software visualizations that are more difficult to classify. One set of these is heap visualizations such as that provided by FIELD where the system displays a picture of how heap memory is being used, using colors to encode information about each allocation and its use. Another set involves relating information to large bodies of source code as in Seesoft [5].

Software visualization is also related to the more general field of information visualization. This is a problem that has received considerable attention in recent years both for visualizing graphs and hierarchical structures and for displays for data mining. Significant work in developing information visualizations has been done at Xerox PARC [13,26], SGI, and elsewhere. We make use of this work by incorporating the visual metaphors that are incorporated into our generic back end visualization engine. The Visage system represents a more general approach that, like ours, attempts to combine data specification, browsing, and visualization in an easy-to-use framework [27]. Our system differs in providing a more data-centric approach, in concentrating more on abstract data from a variety of sources rather than more concrete visualizations from an object data base, and on managing dynamic as well as static data.

6 IMPLEMENTATION AND EXPERIENCE

Cacti is currently implemented as part of the Desert software development environment. It uses the Desert cross reference databases as its data sources. The front end, shown in Figure 6, allows the user to easily define visualizations by defining the data to be used and then selecting and parameterizing the visualization strategy to be used. At this point Cacti generates a file describing the visualization and runs a back end, Mirage, with this file. Mirage is responsible for reading in the data and providing the actual visualization. Cacti and Mirage are implement in about 23,000 lines of C++.

This two-step approach and the use of resource files throughout the process provides a great deal of flexibility. We have extended the resource file definitions for Cacti to allow the definition of classes as well as data sources. This allows Cacti to provide a wide range of predefined data visualizations and simplifies the user's task. In addition, definitions themselves can be saved in such a resource file. The visualization description file output by Cacti for Mirage can also be saved and reused. This allows the system to be used to defined fixed visualizations while still offering the user the flexibility of parameterizing the result.

While our experiences with Cacti and Mirage have been limited (like the overall Desert project, these packages are early prototypes and are not widely used), we are optimistic that this approach will be successful. We have been able to define a wide range of visualizations using the front end. These have been defined quickly and logically. The heuristic algorithm for converting the class and member definitions into a query has worked quickly and has generated the right query for all our tests.

At the same time, there is a significant amount of work remaining to be done on the system. We have to work on making available and integrating more and varied data sources to Cacti to verify that different types of sources will fit comfortably into our model and to make the resultant visualizations more useful. We need to examine a much wider range of different visualizations to verify the heuristics and weights that are used in converting the data definition into a query. We have to build up a library of predefined data classes that are useful to visualize. Finally, we have to offer this facility to a broader audience to get feedback on the user interface and the ease of defining visualizations.

The back end, Mirage, also requires significant work. We need to add more sophisticated mappings between data objects and visualizations, allowing a data object to map into multiple visual objects (e.g. an arc and the node the arc points to). We need to provide user feedback about what object the mouse is pointing to in a meaningful way (the hooks for this are there, but we haven't determine how or what to output as a result). We also need to work on performance issues when displaying large numbers of objects.

7 REFERENCES

1. Bill Appelbe, Kevin Smith, and Charlie McDowell, "Start/Pat: a parallel- programming toolkit," *IEEE Software* Vol. **6**(4) pp. 29-38 (July 1989).

2. Marc H. Brown and Robert Sedgewick, "Techniques for algorithm animation," *IEEE Software* Vol. **2**(1) pp. 28-39 (1985).

3. Marc H. Brown and Marc A. Nojork, "Algorithm animation using 3D interactive graphics," DEC Systems Research Center (1992).

4. I. F. Cruz, "DOODLE: a visual language for object-oriented databases," *ACM SIGMON Intl. Conf. on Management of Data*, pp. 71-80 (1992).

5. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr., "Seesoft - a tool for visualizing software," AT&T Bell Laboratories (1991).

6. K. Goldman, S. Goldman, P. Kanellakis, and S. Zdonik, "Isis: interface for a semantic information system," *Proceedings of the ACM SIGMOD*, (1985).

7. Vincent A. Guarna, Jr., Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur, "Faust: an integrated environment for parallel programming," *IEEE Software* Vol. **6**(4) pp. 20-27 (July 1989).

8. Reed Hastings and Bob Joyce, "Purify: fast detection of memory leaks and access errors," *Proc. Winter Usenix Conf*, (January 1992).

9. P. Honeyman, R. E. Ladner, and M. Yannakakis, "Testing the universal instance assumption," *Information Processing Letters* Vol. **10**(1) pp. 14-19 (February 1980).

10. S. C. Johnson, "Postloading for fun and profit," *USENIX Winter '90*, pp. 325-330 (1990).

11. James R. Larus, "Abstract execution: a technique for efficiently tracing programs," U. Wisc.-Madison Computer Sci. Dept. TR 912 (February 1990).

12. Ralph L. London and Robert A. Duisberg, "Animating programs using Smalltalk," *IEEE Computer* Vol. **18**(8) pp. 61-71 (August 1985).

13. Jock D. Mackinlay, George G. Robertson, and Stuart K. Card, "The perspective wall: detail and context smoothly integrated," *Proc. CHI'91*, pp. 173-179 (April 1991).

14. Scott Meyers and Steven P. Reiss, "An empirical study of multiple-view software development," *Software Engineering Notes* Vol. **17**(5) pp. 47-57 (December 1992).

15. Thomas G. Moher, "PROVIDE: a process visualization and debugging environment," *IEEE Trans. Soft. Eng.* Vol. **14**(6) pp. 849-857 (June 1988).

16. H. A. Muller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji, "A reverse engineering environment based on spatial and visual software interconnection models," *Software Engineering Notices* Vol. **17**(5) pp. 88-98 (December 1992).

17. Brad A. Myers, "Incense: a system for displaying data structures," *Computer Graphics* Vol. **17**(3) pp. 115-125 (July 1983).

18. B. A. Price, I. S. Small, and R. M. Baecker, "A taxonomy of software visualization," *Journal of Visual Languages* Vol. **4**(3) pp. 211-266 (Dec. 1993).

19. Steven P. Reiss, "Working in the Garden environment for conceptual programming," *IEEE Software* Vol. **4**(6) pp. 16-27 (November 1987).

20. Steven P. Reiss, "Trace-based debugging," *Proc. AADEBUG '93*, (May 1993).

21. Steven P. Reiss, *FIELD: A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).

22. Steven P. Reiss, "An engine for the 3D visualization of program information," *Journal of Visual Languages*, (December, 1995).

23. Steven P. Reiss, "Simplifying data integration: the design of the Desert software development environment," *Proc. 18th Intl Conf on Software Engineering*, pp. 398-407 (March, 1996).

24. Steven P. Reiss, "Software tools and environments," in *Software Visualization: Programming as a Multimedia Experience*, ed. Blaine Price,MIT Press (1997).

25. Steven P. Reiss, "Dynamic management of the Desert program data store," Brown University CS Tech Report (1997).

26. George G. Robertson, Jock D. Mackinlay, and Stuart K. Card, "Cone trees: animated 3D visualizations of hierarchical information," *Proc. CHI'91*, pp. 189-194 (April 1991).

27. Steven F. Roth, Peter Lucas, Jeffrey A. Senn, Cristina C. Gomberg, Michael B. Burks, Phillip J. Stroffolino, John A. Kolojejchick, and Carolyn Dunmire, "Visage: a user interface environment for exploring information," *Proc. Information Visualization*, pp. 3-12 (October 1996).

28. Lawrence Snyder, "Parallel programming and the Poker programming environment," *IEEE Computer*, pp. 27-36 (July 1984).

29. John T. Stasko, "TANGO: a framework and system for algorithm animation," *IEEE Computer* Vol. **23**(9) pp. 27-39 (September 1990).

30. M. M. Zloof, "Query by Example: a data base language," *IBM Systems J.* Vol. **16**(4) pp. 324-343 (1977).