To appear in IEEE Symposium on Information Visualization (InfoVis '98), October 1998.

Traversal-based Visualization of Data Structures*

Jeffrey L. Korn
Andrew W. Appel

Department of Computer Science, Princeton University†

July 16, 1998

Abstract

Algorithm animation systems and graphical debuggers perform the task of translating program state into visual representations. While algorithm animations typically rely on user augmented source code to produce visualizations, debuggers make use of symbolic information in the target program. As a result, visualizations produced by debuggers often lack important semantic content, making them inferior to algorithm animation systems. This paper presents a method to provide higher-level, more informative visualizations in a debugger using a technique called *traversal-based visualization*. The debugger traverses a data structure using a set of user-supplied patterns to identify parts of the data structure to be drawn a similar way. A declarative language is used to specify the patterns and the actions to take when the patterns are encountered. Alternatively, the user can construct traversal specifications through a graphical user interface to the declarative language. Furthermore, the debugger supports modification of data. Changes made to the on-screen representation are reflected in the underlying data.

CR Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation - Viewing Algorithms; D.1.7 [**Programming Techinques**]: Visual Programming; D.2.5 [**Software Engineering**]: Testing and Debugging - Debugging aids.

1 INTRODUCTION

Software visualization enables users to mentally picture a computer program or algorithm. It is most often used in algorithm animation systems, where the intended purpose of the visualization is to communicate how an algorithm works. Such visualization is also useful to a debugger, as it can help reveal which parts of the code are not functioning correctly. Some recent PC and UNIX debuggers such as DDD [19] and Deet [5] provide visual representations of data structures. The pictures rendered by these debuggers are essentially mirror images of how the data is laid out in memory. This is in contrast to algorithm animation systems such as Balsa [2] and Zeus [3], where an animator has a finer level of control by hand crafting the pictures through the augmentation of source code with calls to the animation system. Since debuggers usually work with object code, using source-level animations is not practical. Therefore, debuggers aren't able to provide displays that are as abstract and informative as algorithm animation systems.

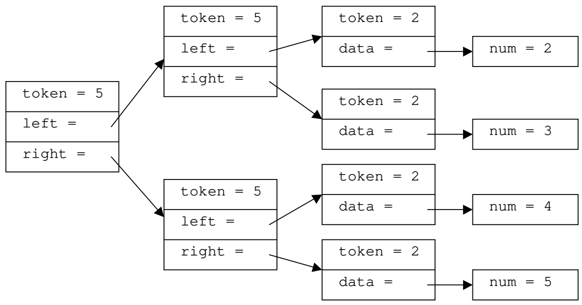

Figure 1: Low-level Data Structure Layout

This paper examines how a debugger can visualize data structures by using externally supplied semantic information to produce useful and informative visualizations. We wish to produce displays that contain more than a picture of what is in the heap. For example, instead of displaying a structure that contains two integer fields as two boxes labeled with the values of the fields, we may wish to represent the data as a point on a two dimensional graph. A data structure that contains an index into an array might wish to draw the contents of that array element rather than simply the index number. The emphasis in this paper is in the methods of collecting the information to be visualized rather than on the pictures produced. Once we have gathered the data, we use a set of standard visualization techniques to display the structures.

Figure 1 shows an example of a tree-like data structure as visualized without any external information. Nodes contain a field named `token` which is an integer. For this data structure, `token` represents either an operation type or a value type, and we'd like to see the operation represented in a meaningful way. By applying some transformation to the structure, we could end up with a drawing like the one shown in Figure 2, which is both more concise and easier to understand.

In order to produce such visualization, there must be a way for the user to specify what the underlying data means. The PROVIDE debugger [7] was one of the earliest debuggers to add a level of abstraction to the visual display of data. With PROVIDE, users could select from a limited set of simple mappings that could display elements in the form of pie charts, bar graphs, etc.

More recent systems have attempted a more flexible and generalizable approach. The Lens debugger [8] attempts to use the techniques found in algorithm animation systems and apply them at the debugger level. Algorithm animation systems use *interesting events* for visualization, which identify key steps in the algorithm where visualization code is inserted. Lens allows the user to at-

*This work is supported in part by AASERT grant DAA G55-97-0209, DARPA order number E381, and NSF grant ASC-9612756.

†Author's Address: 35 Olden St., Princeton, NJ 08544; email: {jlk,appel}@cs.princeton.edu

©1998 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

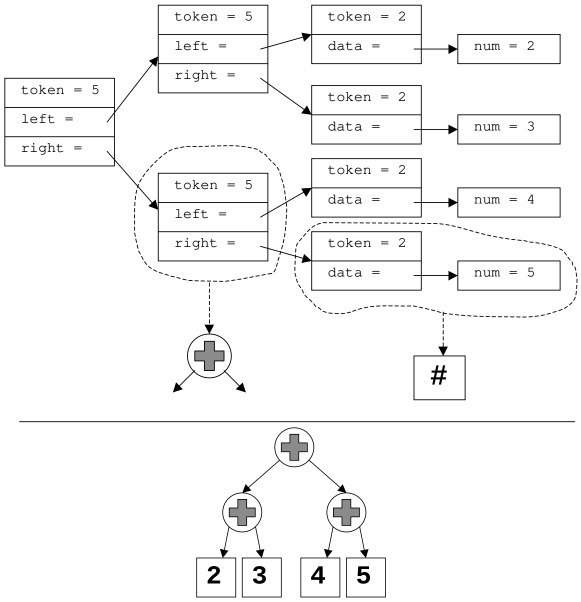

Figure 2: Transforming traditional layout into a more abstract representation

tach algorithm animation instructions to breakpoints, bringing the interesting events model to the level of the debugger. A problem with interesting events is that it is often difficult to identify the appropriate segments of the code to annotate. The construction of a data structure may be sprinkled across the code, resulting in many related but separate annotations. Another problem with interesting events is that they cannot be used to debug an already running program that has created data structures prior to being debugged.

An alternative approach to interesting events, introduced by Roman [14], is called *declarative visualization*. Declarative visualization is a method that defines mappings from program state to graphical objects. It depends solely on data, eliminating the need to know about a program's control flow. Thus, declarative visualization does not need access to the source code of the algorithm being debugged. A system called Pavane [13] uses the technique of declarative visualization. However, Pavane is not a debugging system — it is used to visualize concurrent computations. Its mappings resemble predicate logic and visualization of a mapping requires an access to every object in the heap. This is unsuitable for a debugger which deals with large programs and should only access objects on a need-to-know basis. Pavane also does not allow visualizations to be changed at run time, nor does it allow modification of data. Such features are of use to a debugger.

We present another approach, which we call *traversal-based visualization*. With traversal-based visualization, we take a root object or set of objects and traverse the objects by following any other objects referenced by or associated with the root. As we traverse, we apply a set of predicates to the data that decide how the data is to be drawn. Traversal-based visualization, like declarative visualization, does not require finding and annotating events in program code. Unlike declarative visualization, it allows objects to be examined efficiently by using objects as they are needed, making it possible to use in a debugger. Traversal-based visualization is particularly useful for working with tree-like data structures. However, it works with any linked structure, even those with cycles.

This paper presents a debugger built using traversal-based visualization. Using a user-supplied specification of visual mappings, it traverses the target's data structures to produce abstract displays. The predicates take the form of patterns, which are more efficient than generic predicates. The system is designed to be capable of working with a variety of languages, including C, C++, and Java. The current implementation only supports Java, and although this paper will focus on Java, it is important to note that the techniques are applicable to other languages as well.

The system allows visual mappings to be changed at run time and provides a mechanism to add callbacks that allow changes made to on-screen objects to be reflected in the underlying data. Mappings can be constructed with a textual specification, or through a user interface which allows users to quickly put together useful displays.

The remainder of this document presents an overview of the system. We give examples of how the system can be used to visualize some sample data structures, and discuss how the system is implemented. We conclude by summarizing the current status and future directions of this work.

2 FOUNDATION

The goal of this work is to produce a debugging environment where data structures can be abstractly displayed. This paper does not go into the details of the fundamental operations that the debugger provides such as stepping and setting breakpoints (see [5]), but instead focuses only on the visual display of values. Which objects are displayed and when they are displayed are something that the debugger needs to handle, but the discussion here focuses on *how* they are displayed. We will assume we are given an object or set of objects to draw, and we only have access to symbolic information of the object code.

The guiding principles that were considered in the design of the visualization system are as follows:

1. **Declarative specification:** Visualizations should be specified through a declarative language to provide maximum flexibility. Such a language makes it possible to write tools that automatically generate specifications.

2. **Visual interface:** A casual user should not have to learn a new language to put together a simple visualization. Therefore, a visual interface to the declarative language should be provided for simplicity.

3. **Data Modification:** There should be a way to modify the underlying data in the program by modifying elements of the visualization.

This section gives an overview of the system that accomplishes these goals. The specification language consists of a set of *rules* that define how to draw objects matching a given structure. Each rule defines a *pattern*, which specifies the form that a value must match, and a set of *actions* that create entities to be displayed. The entities are rendered by a separate component. Figure 3 shows how each of the components interact. The remainder of this section further describes these components.

Data Model

First, we must make some assumptions about the underlying data. We assume that our data consists of objects, where each object is an instance of a given type, and an object may be a reference (pointer) to another object. An object type is either a primitive type (integer, string, etc.) or made up from a collection of other types (accounting for classes, structures, arrays, etc.). This assumption is sufficient for visualization in languages such as C, C++, Java, and Modula-3. It may be less so for languages that do not have types or support mutation, such as scripting languages or ML, which are beyond the scope of this paper.

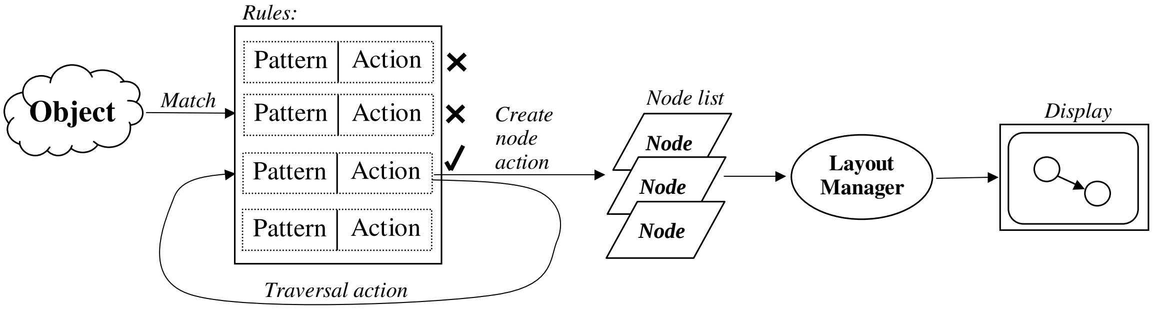

Figure 3: System Components

Patterns

A pattern defines a set of structures that fit a given criteria, where the form of the structure is specified by constraining values or supplying wildcards. Patterns are to data structures what regular expressions are to strings. We will explain patterns by showing their grammar:

```
pattern        =  type-name label [ pattern-body ]
                  [ when-clause ]

pattern-body   =  pattern-body "&&" pattern-body
               |  pattern-body "||" pattern-body
               |  sub-pattern

sub-pattern    =  simple-pattern
               |  struct-pattern

simple-pattern =  relation expression

struct-pattern =  "=" "{" { field ";" } "}"

field          =  type-name field-name [sub-pattern]

when-clause    =  "when" "(" expression ")"

relation       =  "=" | "!=" | "<" | ">" | ...
```

Figure 4: EBNF Grammar for Patterns

Each pattern is defined with a *type-name* and *label*, where the *type-name* is the type of the object to match the pattern against, and the *label* is a name for the pattern. Optionally, a *pattern-body* can be declared to further constrain the pattern. If *pattern-body* is not specified, the pattern is a wildcard for the given type.

A *pattern-body* is a boolean expression of *sub-pattern*s, where `&&` is used to match both *pattern-body* fields, and `||` is used to match either.

There are two basic types of *sub-pattern*s. First, a *simple-pattern* is a pattern that matches a primitive type in the language such as an integer or string. We use the *relation* to compare the object to *expression*. *relation* is `=`, `!=` or a comparison function (eg. <) for numerical values. The *expression* can be any source-level expression, which is evaluated once when the pattern is defined. For example, the pattern to match any integer is:

```
int x
```

The pattern to match any integer greater than zero is:

```
int x > 0
```

Second, a *struct-pattern* matches an object that contains a collection of fields, such as a class in Java or a struct in C. A *struct-pattern* specifies a list of patterns that are used to match fields of the structure. Any field not listed is unconstrained (a wildcard). If we have a class `Point` with two elements, `x` and `y`, the pattern to match any instance of `Point` is:

```
Point p
```

The pattern to match an instance of `Point` where the field `y` is non-zero:

```
Point p = {
    int x;
    int y != 0;
}
```

Further nesting is possible. For example, If `Point` were contained within another class `Element`, we could use a pattern such as the following:

```
Element e = {
    Point point = {
        int y != 0;
    }
    UserData data != null;
}
```

The optional *when-clause* of a pattern can be used to specify a condition that must also be satisfied when a pattern matches an object. Unlike the *expression* field in a *simple-pattern*, which is evaluated once when the pattern is defined, the *expression* of the *when-clause* is evaluated each time an object is matched against the pattern. Thus, a *when-clause* provides a way to allow general predicates, though at additional cost. For example, a *when-clause* could be used as follows:

```
Point point = {
    int y > 0;
} when (point.x < Math.sqrt(point.y))
```

The use of patterns makes it possible to perform efficient visualizations of data. The set of patterns in a specification can be compiled into a finite automaton, in effect making it possible to avoid matching each pattern individually. This is similar to the way regular expression matchers work. If we have a large pattern specification, there will be minimal slowdown during the matching phase after the specification is compiled to the automata. Once an object is matched with the automaton, it will then evaluate any *when-clause* expressions. If multiple patterns match an object, the pattern defined first will be returned as the match.

Actions

Along with a pattern, one must specify what to do when an object is encountered that matches the pattern. This typically involves the creation of *nodes*, which are entities that are to appear on screen. Nodes are represented with a set of attributes, which are (*name*, *value*) pairs describing the node. For example, a node could contain attributes for its color, shape, label, and font. The actual rendering of the node, described in the next section, is done separately. Actions can set attributes for a node using expressions that include the objects from the matching pattern. Alternatively, an external function can be called to set the attributes.

Actions may also specify other elements of the data structure that need to be drawn. For example, a pattern that matches a node in a tree could request that its children also be drawn. An integer field in a data structure which represents an index into an array could request that the appropriate element of the array be drawn. Figure 5 shows the grammar for actions.

Environments

When traversing a data structure, it is often useful to pass along state information from an element of the structure to its descendants. For this purpose, we use *environments*. Environments maintain a set of variables and values for a particular pattern. Environments are inherited from the pattern matching an object by the patterns matching subsequently traversed objects. The values of environment variables are set in the actions of a rule and are arbitrary expressions that can be computed using objects in the target program.

When an object matches a pattern, an environment variable is created that can be used to reference the object that matched the pattern. Its name is the identifier supplied as *label* in a pattern definition. When an action or when-clause refers to a symbol in an *expression*, the environment is first checked for the symbol. If the symbol is not found in the environment, global variables in the target are checked.

Layouts

Once a set of nodes is constructed from a traversal of a data structure, it is sent to a particular layout manager to be drawn. Layout managers draw the nodes by looking at their attributes. For example, if a data structure represents a set of two dimensional points, a traversal could produce a set of nodes with attributes named *x* and *y*. A layout manager drawing points on a two dimensional plot would use this list of nodes to render points based on these attributes.

Currently, the system contains layout managers for directed graphs, hierarchical lists, and two dimensional plots, with more are to be added. Each of these layout managers use standard visualization techniques, so we do not go into the details of layout in this paper. Users are also permitted to add layout managers of their own which can be reused across multiple visualizations.

3 EXAMPLES

To best illustrate how the visualization system works, we will look at some examples. The examples consider the debugging of a tree-like data structure in Java. In our data structure, each element of the tree is either an integer or a binary operation. Each element is a subclass of the class `Expr`. Our class definitions are as follows:

action-list	=	*action* { "," *action*}
action	=	(*node-creation* | *traversal*) *val-list*
node-creation	=	[*environment-var* =] *node-name*
traversal	=	"->" *expression*
val-list	=	"(" { *identifier* "=" *expression* } ")"

Figure 5: EBNF Grammar for Actions

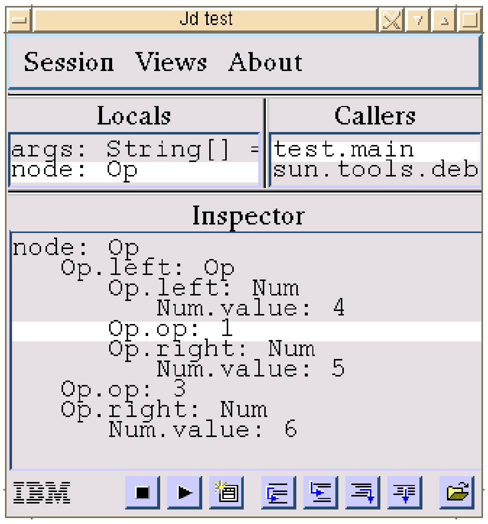

Figure 6: Traditional Layout

```
public abstract class Expr { }

public class Num extends Expr {
    private int value;

    // Methods not shown
}

public class Op extends Expr {
    final static int PLUS = 1;
    final static int MINUS = 2;
    final static int TIMES = 3;
    final static int DIV = 4;

    private Expr left;
    private int op;
    private Expr right;

    // Methods not shown
}
```

The class `Op` has an integer field indicating the operation type. Suppose we are at a breakpoint and we wish to graphically display a tree of type `Expr`. Figure 6 shows how a typical graphical debugger might draw a tree instance without user input [9]. In fact, some existing debuggers are incapable of drawing this much. Since `left` and `right` are declared as type `Expr` but instances are always of one of `Expr`'s subtypes, some debuggers will draw the fields as an `Expr` which contains no fields.

A major problem with the drawing in Figure 6 is that the `op` field is shown as a number, making it difficult to see what kind of operation the node actually represents. There is no information in our data structure that says that the `op` field should be interpreted as one of the defined constants in the class `Expr` instead of a plain integer. Thus, we will write a set of patterns with actions to display the operation field more appropriately:

```
Num numPattern : node=TreeNode(label=numPattern.value),
                 TreeEdge(from=parent, to=node);

Op plusPattern = {
    int op = Op.PLUS;
} : node=TreeNode(icon="plus.xbm"),
```

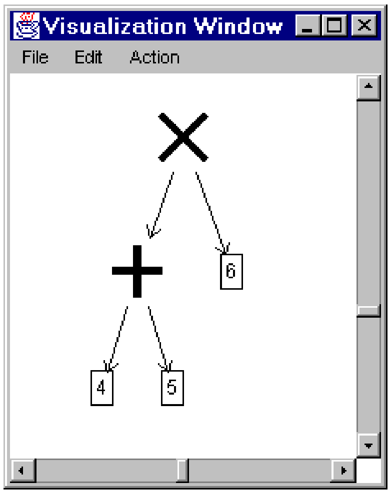

Figure 7: Layout with Patterns

```
    TreeEdge(from=parent, to=node),
    -> plusPattern.left(parent=node),
    -> plusPattern.right(parent=node);

Op timesPattern = {
    int op = Op.TIMES;
} : node=TreeNode(icon="times.xbm"),
    TreeEdge(from=parent, to=node),
    -> timesPattern.left(parent=node),
    -> timesPattern.right(parent=node);

// etc..
```

These rules create specialized `Op` nodes depending on the value of the `op` field. When the nodes are drawn, they will be viewed as seen in Figure 7. The pattern named `plusPattern` matches any object of type `Op` which has its `op` field set to `Op.PLUS` (which is statically defined to be 1). If an object matches this pattern, then the action taken is to create two nodes, one of type `TreeNode` and one of type `TreeEdge`. Attributes may be specified when a node is created. In this case, we specify the `icon` attribute for `TreeNode` to contain a reference to a plus icon. The `TreeNode` node has other attributes such as color and font, but we do not need to specify attributes for which we use the default value. The `TreeEdge` node connects two nodes together. It identifies the two nodes using the environment variable `parent`, which has been passed down from the parent object, and `node`, which is defined when the `TreeNode` is created.

For each of the rules, we also specify other objects that are to be traversed. In each of the patterns in this example, we draw the `left` and `right` fields. The specification of these fields will permit the entire tree to be traversed. We pass down the environment variable `parent` set to the node created in the current rule so that when the object's fields are traversed, they can use the `parent` environment variable to properly link their nodes to the tree.

To demonstrate the use of *when-clauses*, let us consider the following example, in which we modify our patterns above to highlight any integers that are contained below a `TIMES` node when one of the operands is 0.

```
Num redNumPattern when (hilite == 1)
  : node=TreeNode(label=redNumPattern.value, color="red"),
    TreeEdge(from=parent, to=node);

Num greenNumPattern when (hilite != 1)
  : node=TreeNode(label=greenNumPattern.value,
        color="green"),
     TreeEdge(from=parent, to=node);

Op times0Pattern = {
    int op = Op.TIMES;
    Num left = {
        int value = 0;
    }
} || {
    int op = Op.TIMES;
    Num right = {
        int value = 0;
    }
} : node=TreeNode(icon="times.xbm"),
  TreeEdge(from=parent, to=node),
  -> times0Pattern.left(parent=node, hilite=1),
  -> times0Pattern.right(parent=node, hilite=1);

// Previous rules for "Op"
```

Above, when an `Op` node is reached that has its `op` field set to `Op.TIMES` and one of its operands is zero, it performs the same actions as the previous example, except it passes down the environment parameter `hilite` a value of 1. The first two patterns above specify preconditions on the environment by using *when-clauses*. The first pattern is only applied if the environment has `hilite` set to 1, and the second is applied in all other cases. Thus, if a `Num` node is reached with the `hilite` environment variable set, we know that it has previously been found to be underneath a `Op.TIMES` node where one of the operands is zero. We can then draw it specially, in this case with a distinguishing color.

Note here that this visualization is difficult to do with other models such as interesting events and declarative visualization. Since `Num` instances don't contain a reference back to the parent, determining such a property is not possible without going through the data structure and keeping around information as it is traversed. Our experience has found that environment variables are frequently useful for producing visualizations of data structures.

Visualization of Real Programs

Using the traversal specification language, we have constructed visualizations for some existing programs. In Figure 8, we see the visualization of a program that finds the longest path through a directed graph with weighted edges. The visualization is produced by traversing a linked list of edges and nodes, turning each into an object that is sent to the directed graph layout manager. The layout manager chooses the location for the nodes. Edges that appear in the longest path are determined by checking the value of a field named `inGraph`, so we define patterns to draw such edges with a dashed line. The pattern specification is as follows:

```
Graph graph = {
        Node nodeList;
        Edge edgeList;
} : -> graph.nodeList,
    -> graph.edgeList;

Node node : DagNode(label=node.num, id=node),
            -> node.next;

Edge edge = {
        int inGraph = 1;
} : DagEdge(label=edge.weight, style="dashed",
       from=edge.left, to=edge.right),
    -> edge.next;

Edge edge = {
        int inGraph = 0;
} : DagEdge(label=edge.weight, from=edge.left, to=edge.right),
    -> edge.next;
```

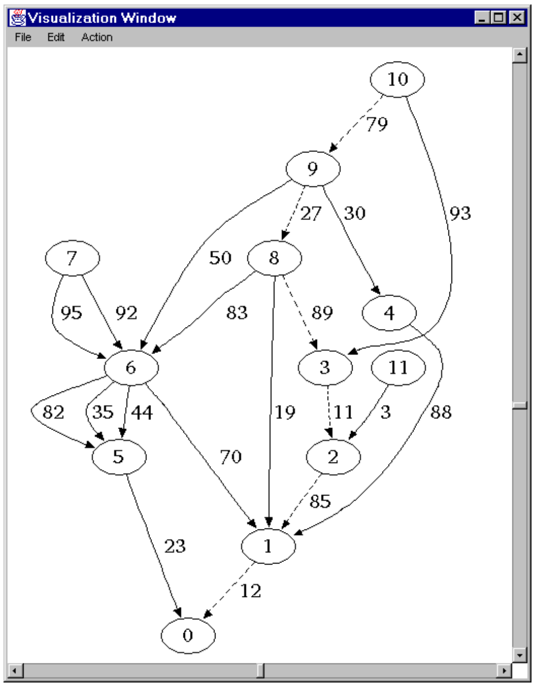

Figure 8: Visualization of Longest Path

In Figure 9, we see a visualization of an algorithm that computes a convex hull from a list of points. The points are stored in an array, and the set of points found to compose the hull is stored separately in a stack. To produce the visualization, we traverse the array of points, turning each point into a node with x and y coordinates, and then the stack. The traversal of the stack creates line segments using successive elements in the stack as end points. The specification (not shown here) is around 30 lines long.

Automated Pattern Generation

An advantage of using a declarative language is that it is possible to write tools that automatically generate specifications.

Generated specifications are used to produce simple visualization of data structures similar to that of a typical visual debugger. Information in the target's symbol table is used to create the specifications. For each type in the symbol table, a pattern is created to handle the rendering of that type. The actions of the rule are to create a node for the object, displaying each of the fields of the object in the node. Any fields that are references are specified to be further traversed. The nodes created by these patterns contain attributes that can be visualized by either the hierarchical list or directed graph layout managers. The automated pattern specification is useful to a user who wants to produce basic displays without writing patterns. It should not be necessary to write specifications that can be created automatically.

Generated specifications can also be useful for other applications. Consider a tool such as *lex* or *yacc*, which takes a high-level specification (e.g. lexer or parser) and generates lower-level code. Debugging the generated code might be difficult, as the correspondence between the abstractions in the input file and the generated data structures may not be easy to determine. If we augment such tools to additionally generate a pattern specification, we can use the patterns to assist in the debugging process. This type of generation of external information to be used by the debugger is similar to what is already done by compilers, as they emit extra symbol table information when appropriate flags are specified.

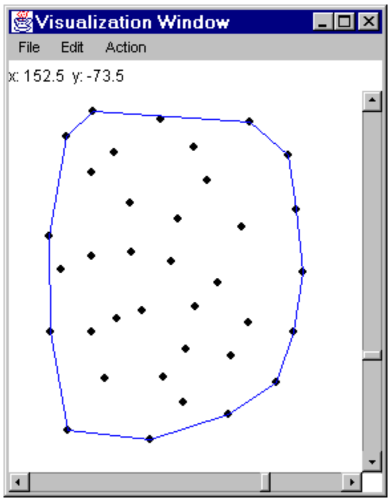

Figure 9: Visualization of Convex Hull

4 VISUAL PATTERN CONSTRUCTOR

Our debugging tool also provides a visual interface to the pattern language, making it possible to construct visualizations entirely through a user interface. This is especially useful for first-time or casual users of the system who don't wish to learn the pattern language. The visual interface allows the patterns and actions to be constructed from scratch or by starting with an example.

The first step to creating a pattern from scratch is to create a root pattern node. A root node is created by selecting from a list of known types (extracted from the symbol table). The root node then appears on screen, representing the pattern that matches any object of the specified type. The user can then use a pop-up menu to further constrain the pattern. If the type is a structural type, the menu lets the user choose from the the set of fields defined in the structure. For primitive types, one of "$=$", "$!=$", "$>$", "$>=$", "$<$" and "$<=$" can be chosen to specify a value to match. A user can also select "unrestricted", which removes any existing submatches and turns the pattern into a wildcard. Figure 10 shows the user interface of the pattern constructor.

Once a pattern has been visually specified, a set of actions can be defined through the interface. Actions are either nodes to be created or other objects to be traversed. For actions that create nodes, an interface is presented through which attributes for the node can be entered. Similarly for traversal actions, an interface exists to set passed down environment variables.

Patterns can also be constructed by selecting a group of nodes that have been already rendered on the display, similar to what is shown in Figure 2. The selected nodes become an exact pattern match for that substructure. The pattern can then be further refined (by changing a subpattern to be "unrestricted", adding fields, etc.) using the pattern editor interface.

5 SYSTEM IMPLEMENTATION

This section gives an overview of how the system is implemented. The current implementation is approximately 9,000 lines of Java code.

Debugger Interaction

Our system uses an abstract interface to access symbol table information through a debugging library (not discussed here, see [5]).

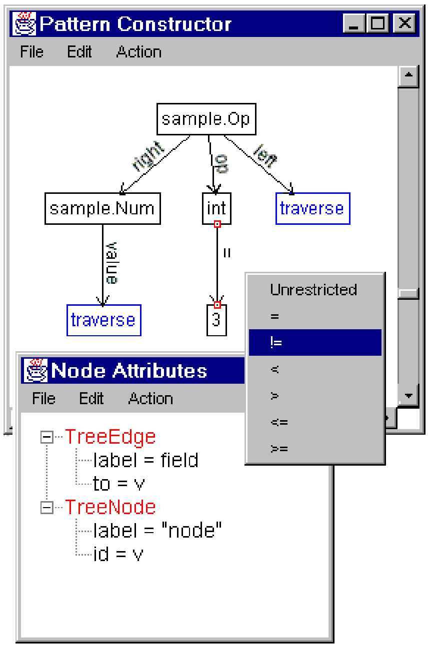

Figure 10: Visual Pattern Constructor UI

An advantage to having an abstract interface to the debugger is that we can switch between alternate implementations of the API. For example, there are two implementations in Java. The first is built on top of Sun's `RemoteDebugger` package, which is used to debug a Java program running in a separate interpreter (the target). The second is built on top of the Java *reflection* API, which is used to gather type and object information in the current interpreter (the debugger). It is useful to switch between accessing objects in the debugger and the target, and the API provides the flexibility to do this.

Updates and Redrawing

It is undesirable to redraw an entire data structure each time part of it is changed. Redrawing requires retraversing every element of the structure, which is inefficient and does not provide the user with information on which parts have changed. Therefore, our system has the ability to determine which on-screen nodes need to be updated based on a set of modified objects in the target.

The debugger is responsible for determining which of the underlying objects in the target have changed and reporting these objects to the visualization component. It is difficult for a debugger to tell when an object has been modified, but it is beyond the scope of this paper. Once the system has determined which objects have changed, it is then necessary to update the display appropriately.

In order to tell which of the on-screen nodes need to be updated, a mapping between in-core objects and on-screen nodes is maintained. This mapping is constructed during object traversal. When a pattern matches an object, each object referenced in the pattern is mapped to the node or nodes created by the actions of that pattern. When an object is redrawn, the system first checks which pattern the object matched before the modification using the mapping table. It then matches the modified object against the patterns, and checks if the newly matched pattern is different than the previous one. If the same pattern is matched, and the object and environment values are equal, no further action needs to be taken.

If a different pattern is matched, all of the old nodes created by that pattern are marked for deletion. The new pattern is applied,

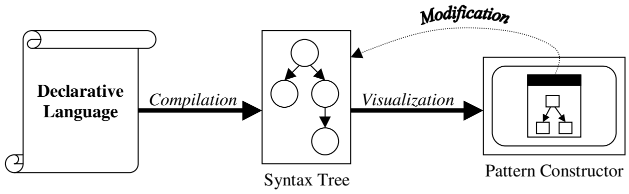

Figure 11: Visual Pattern Constructor Design

creating a new set of nodes. We then look at the dependencies of the new pattern. For each dependency denoting a different object than in the old pattern, we delete the old dependency, and traverse the new dependency.

After a modification is applied, the layout manager receives a list of nodes to be created and deleted. The layout manager is then responsible for updating the display based on the values of the new nodes.

Modification

Now we examine how modification to an object's on-screen representation is propagated to the underlying data structure. The layout manager is responsible for handling user interaction with displayed nodes. When a node is manipulated, the layout manager reflects the changes into the object by changing its attributes. For example, if a point is dragged in a two dimensional plot, the interface would update its `x` and `y` attributes. Attribute changes may be monitored by specifying a callback routine that is invoked when a node is modified.

In the callback routine, it is possible to use the modified attributes to make changes to the underlying object. The routine may use node attributes and environment variables to determine what needs to be changed and how. A callback can also do a reverse-mapping to determine which object or objects were used to synthesize attributes of the node through a set of library functions. The callback uses the debugger interface to make changes to the underlying objects. Changes are then sent to the visualization component, where the appropriate nodes are redrawn as described previously in this section.

If no callbacks exist to monitor node attributes, then the on-screen nodes will be changed but the underlying object will not be modified.

Visual Pattern Constructor

The visual pattern constructor is itself a traversal-based visualization. In the declarative language, patterns are parsed into an abstract-syntax tree before they are internally processed. The visual pattern constructor, which provides a user interface to the declarative language, is written with a pattern specification to visualize this abstract-syntax tree. As patterns are edited on the display, the abstract-syntax tree is manipulated to reflect the changes (through callback functions).

Figure 11 shows the design of the pattern constructor. One difference between a typical visualization and the visualization the pattern constructor uses is that the objects the pattern constructor visualizes are in the address space of the debugger instead of the address space of the target. Thus, we use an alternate implementation of the debugging functions (the Reflection API) to access the underlying objects.

6 DISCUSSION

Preliminary results from the system have been promising. We have used it to put together visualizations for a project called *Zephyr* [18]. Zephyr provides a language for describing tree-like intermediate forms in compilers (for example, abstract-syntax trees), and includes a tool that generates data structures matching the descriptions. Our visualizations draw the intermediate forms graphically. The use of patterns makes it possible to easily distinguish between different types of nodes, as well as identify and simplify commonly used idioms. Our experience with traversal-based visualization has led to improved displays while requiring little work to produce them.

Another application built from the declarative language was the user interface to generate the declarative language itself. Through putting together the visual pattern constructor, we were able to make use of the support for modification.

Our experience with the system is in early stages. So far, we have had success using the debugger in the compiler domain as well as with a handful of small programs, but the next step will be to explore other areas. There are many commonly used visualizations that we would like to implement, such as sorting algorithms. We plan to produce visualizations for the set of algorithms typically used as examples in algorithm animation systems.

One of the primary goals of this work is to make it possible for novice users to quickly and effortlessly construct useful visualizations. To accomplish this, we will further pursue the development of user interfaces to the declarative language, as well as write tools that automatically generate specifications for various applications.

Although we do not have performance numbers at this point, we expect that our system will scale well. By compiling patterns into an automata, it is possible to handle a large number of rules in an efficient manner. We would like to see how well traversal-based visualization works in large applications.

The current implementation only provides static pictures. We hope to provide smooth animations found in algorithm animation systems in the future. In order to do this, it is necessary to know the relationship between the nodes in successive steps of an algorithm. For example, if we are animating a sorting algorithm, we need to distinguish between setting two elements in array to new values and swapping two elements. Thus, we need to provide a way to specify such supplementary information to layout managers, similar to what is done in algorithm animation systems.

Support for other programming languages besides Java is upcoming. This will make it possible to evaluate how well our data model and patterns work for languages such as C and C++.

7 CONCLUSION

We have introduced a new model of software visualization called *traversal-based visualization*, which is capable of displaying abstract representations of data structures in a debugger. Traversal-based visualization makes it possible to write a specification of patterns and actions that provide the semantic information needed to draw objects in an informative way. We have implemented a debugger that allows transformations to be specified with a pattern-based language. The debugger also provides a user interface to this language. The system is functional for Java, and development is ongoing. See the author's home page at `http://www.cs.princeton.edu/~jlk/viz` for more information.

References

[1] R. A. Baeza-Yates, L. Jara, G. Quezada. VCC: Automatic Animations of C Programs *Proceedings of Compugraphics*, December 1992, 389–397.

[2] M. Brown. Exploring Algorithms using Balsa-II *IEEE Computer*, 21(5):14–36, May 1988.

[3] M. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing *IEEE Workshop on Visual Languages*, Kobe, Japan, 1991, 4–9.

[4] R. A. Duisberg. Visual Programming of Program Visualizations: A Gestural Interface for Animating Algorithms. *IEEE Workshop on Visual Languages*, Washington, D.C., 1987, 55–66.

[5] D. R. Hanson, J. L. Korn. A Simple and Extensible Graphical Debugger *Proceedings of the Winter USENIX Technical Conference*, Anaheim, CA, January 1997, 173–184.

[6] C. McCreary. The VGJ Graph Drawing Tool `http://www.eng.auburn.edu/department/cse/research/graph_drawing/vgj.html`.

[7] T. Moher. PROVIDE: A Process Visualization and Debugging Environment *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[8] S. Mukherjea, J. T. Stasko. Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-Level Debugger *ACM Transactions on Computer-Human Interaction*, 1(3):215-244, September 1994.

[9] D. Lieber. The Jikes Debugger `http://www.alphaworks.ibm.com/formula/jikesdebugger`.

[10] B. Myers. A System for Displaying Data Structures *Computer Graphics*, 17(3):115–125, July 1983.

[11] S. North, E. Koutsofios. Applications of Graph Visualization *Proceedings of Graphics Interface*, 1994, 235–245.

[12] S. P. Reiss, S. Meyers, C. Duby. Using GELO to Visualize Software Systems *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, 1989, 149–157.

[13] G.-C. Roman, K. Cox, C. Wilcox, J. Plun. Pavane: A System for Declarative Visualization of Concurrent Computations *Journal of Visual Languages and Computing*, 3(1), January 1992, 161–193.

[14] G.-C. Roman, K. Cox. A Declarative Approach to Visualizing Concurrent Computations *Computer*, 22(10):25–36, October 1989.

[15] T. Shimomura, S. Isoda. Linked-List Visualization for Debugging *IEEE Software*, 8(3):44–51, May 1991.

[16] J. Stasko, J. Domingue, M. Brown, B. Price, editors. *Software Visualization* MIT Press, February, 1998.

[17] Sun Microsystems. Java Foundation Classes `http://java.sun.com/products/jfc`.

[18] D. C. Wang, A. W. Appel, J. L. Korn and C. S. Serra. The Zephyr Abstract Syntax Description Language *USENIX Conference on Domain-Specific Languages*, Santa Barbara, October 1997.

[19] A. Zeller and D. Lütkehaus. DDD — a free graphical front-end for UNIX debuggers *SIGPLAN Notices*, 31(1):22–27, January 1996.