Abstraction and Visualization in Graphical Debuggers

Jeffrey Lawrence Korn

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

November 1999

Abstract

As computer systems become increasingly pervasive, the size and complexity of software grows. An inevitable consequence of such growth is the presence of software errors, for which half of development time is spent correcting. Debuggers facilitate tracking and correcting errors by increasing the *visibility* of a program. This thesis examines how debuggers can be constructed that enhance visibility of software by presenting programs at a higher level of abstraction. In particular, we describe the construction of three systems. The first system, Deet, addresses problems that arise in the implementation of graphical debuggers. It focuses on aspects of portability, language-independence and complexity. Deet works with a variety of operating systems, languages and architectures by separating machine-dependent pieces of the debugger into a minimal component called a *nub*. The second system, Chava, provides visualization of object code and can display relationships such as class inheritance diagrams, call graphs, and module dependencies. Chava also supports comparison between two versions of a program, and is useful for reverse engineering applications. The third system, TraVis, provides visualization of program data in the debugger. TraVis draws data structures using a technique called traversal-based visualization, in which the user defines how in-core objects are turned into on-screen entities as they are traversed by the debugger. The mappings make it possible to display data at a higher semantic level without modification of source code.

Acknowledgments

First, I would like to thank my advisor, Andrew Appel, for his time, suggestions and guidance. He gave me considerable freedom in my research yet helped point me in the right direction. It has been pleasure to work with him.

I would like to thank my readers, David Dobkin and Stephen North. David Dobkin was extremely accessible and gave solid advice. Stephen North's technical expertise was invaluable. I would also like to thank the other members of my committee, Perry Cook, Robert Sedgewick and Adam Finkelstein for providing good ideas that helped improve this thesis. David Hanson and Anne Rogers were also a great help at Princeton. Dave got me started in my work with debuggers. Anne was particularly helpful with both technical and personal advice.

I am fortunate to have worked at AT&T Labs - Research during some of the summers while I was at Princeton. In particular, collaboration with Yih-Farn Chen and Eleftherios Koutsofios has had significant impact on me. Working with them has been both pleasurable and rewarding.

I would like to thank the friends I have made at Princeton: Pritpal Ahuja, George Chesakov, Wagner Correa, Stef Damianakis, Chris Dunworth, Corrie Elder, Jon Forsyth, Neophytos Michael, Jason Radley, Mukund Raghavachari, Rob Shillner, Kevin Skadron, Kostas Tsioutsiouliklis, George Tzanetakis, Tassos Viglas, and Dan Wang.

Finally, I would like to thank my family (father David, mother Susan, brothers Flip, Adam, and dog Ubu) for all their support during my time in Princeton. I couldn't have made it through without their encouragement and support.

Funding support for this work was provided by National Science Foundation CCR-9625413, DOD Contract MDA904-97-C-0247, National Science Foundation ACI-9612756, and Navy DAAG55-97-1-0209.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our reliance on computer systems is steadily increasing. Software is used practically everywhere, including banks, hospitals, airports and supermarkets. As the use of computer programs becomes more pervasive in society, we demand more functionality from such systems. This results in additional size and complexity of software, and is perpetuated by faster CPUs and increased memory capacities. For example, Windows 95 contains more than 10 million lines of source [BP98]. An inevitable consequence of larger, more complex programs is the presence of errors, since programs don't always work as they should. Bugs are a major part of the software development process, and typically account for more than half of the total cost of a program over its lifetime [Boe81].

There is a need for tools that help the creators and maintainers of software find and remove such errors in their code. The work presented in this thesis contributes to the process of software development by facilitating the discovery of programming errors.

The Software Crisis

The complexity and size of software is exploding. Estimates show that the amount of code in most consumer products doubles every two years [Gib94]. Software is often over budget, behind schedule and unreliable. Glass [Gla98] reports that 89% of projects have schedule overruns, and 55% are over budget. The number of large projects that end up canceled is steadily rising. These problems found in software projects are serious, and are often referred to as the *software crisis*. Dealing with these issues is essential to the field of software engineering.

Examples of failed software systems are plentiful. The Advanced Automation System (AAS) is one such project that fell victim to the crisis. The system, which was built by IBM for the Federal Aviation Administration (FAA), was designed as an air-traffic control system that could handle the dramatically increasing number of commercial flights. The project, started in 1981, cost 3.7 billion dollars and was eventually scrapped in 1994. It ended due to the staggering amount of system bugs. At the time of termination, there were over 3,000 defects reported [Gla98]. Despite detailed plans and careful management, the builders of AAS were unable to foresee the profusion of errors that resulted.

AAS is not an isolated example. Similar situations developed with Denver's airport baggage-handling system and California's Department of Motor Vehicles registration system. Even the most thought out designs can go awry, and errors will occur in systems as long as humans create programs.

Bugs are often a major reason for project failure. Debugging is a difficult phase of software development because it is not possible to determine in advance how many bugs will arise, where they will be located, and what will be required to fix them. They appear late in the process, usually when the schedule is tight, and can therefore be the determining factor in the success or failure of a project.

Typically, large pieces of software are broken down into many components which are developed separately. After the components are written, they are put together. It is at this point where the most difficult problems usually occur. Interactions between modules cannot always be known in advance and often are subtle. It is difficult to find the source of such errors, which motivates the need for tools which help with this process.

Another difficulty with debugging is that a project may have multiple maintainers throughout its lifetime; often, the person debugging a piece of code is not the person who wrote it. This is problematic because the maintainer cannot always tell what the programmer had in mind when the code was written, making it hard both to track down the problem and to fix it correctly. The maintainer is only working with a set of instructions and lacks semantic content that is absent from the code. Comments attempt to overcome missing semantic information, but are often missing, unclear, or don't match the code.

Since bugs dominate development time, debugging tools can have major impact. Microsoft reported addressing 1,200 bugs per week during the debugging phase of Windows NT [CS95]. Fixing the bugs resulted of about 1,000 new bugs each week. Even minor improvements to debuggers can result in significant savings to a project. This thesis contributes in that area.

The Role of Debuggers

This chapter will discuss the role of debuggers in correcting faulty software. The term *debugger* is really a misnomer, since it is the programmer that actually fixes the bug. The

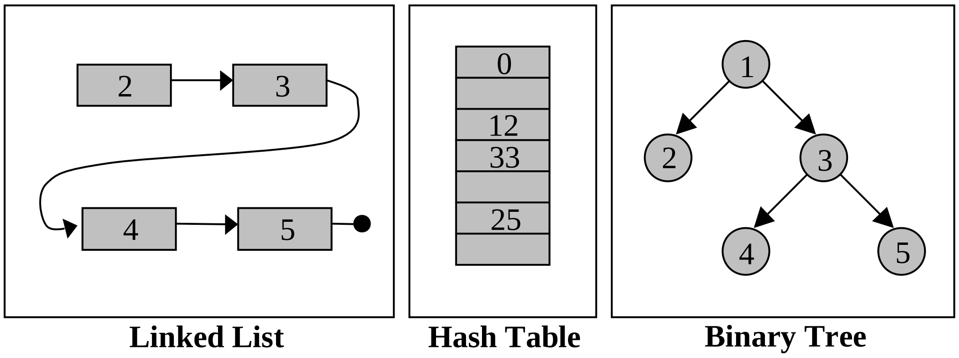


Figure 1.1: Examples of data structure abstractions

debugger's job is to help expose it. We see debuggers as *visibility tools*, tools that make the program visible to the user in a variety of ways. To this end, debuggers also have use beyond the scope of finding errors. They can be used to see how a program or part of a *correctly* functioning program works.

1.1 Bugs in Abstractions

According to Dijkstra, "the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases is called *abstraction*," and patterns of abstraction play "a vital role in the whole process of composing programs." [Dij72]. Examples of abstractions are linked lists, hash tables and binary trees (see Figure 1.1). Unfortunately, most debuggers work at the language level and don't adequately handle such abstractions.

In a study of programming errors in TEX [Knu89], Knuth kept a log of every bug and fix in the system over a ten year period. To analyze the bugs, he devised a classification scheme for errors that were introduced. He distinguished categories of bugs by level of difficulty. One group had categories of errors that were "simple flaws in the basic mechanics of programming." Examples included updating the wrong variable name (*blunder or botch*), forgetting to increment a loop counter (*forgotten function*), misinterpreting operator precedence (*language liability*), calling a function with incorrect arguments (*mismatch between modules*), improperly handling erroneous input (*reinforcement of robustness*), and typos that were not caught by the compiler (*trivial typo*). This group of bugs were typically cases in which the right idea was known when the code was entered, but something was forgotten or written down wrong. Knuth noted that such bugs were easy to fix once identified. A second group of bugs were distinguished as being at a higher level. These were errors in the algorithms and data structures used in the program, or subtle bugs resulting from complex interactions between different parts of the program. Knuth found that these were harder errors to correct than the first group and took more time to fix.

Other studies support Knuth's finding. In Beizer's analysis of approximately seven million statements of code with 11,678 reported bugs [Bei90], 68% of the errors can be classified as high-level programming bugs (those which would fit into the second group of Knuth's breakdown). Of the high-level bugs, 45% were bugs in the definition, structure or use of data.

Debugger Support

Typical debuggers provide little in the way of solving the higher-level problems that arise when coding. For example, detecting a linked list that contains a cycle typically requires printing out each element of the array. There is no single command in a debugger that can be invoked to check this property.

Advancements in debuggers are often focused on tracking down lower-level bugs. For example, tools such as Purify [HJ91] and Insure [KH96] deal with memory management and dynamic storage for C programs. However, the tools don't apply as well to other languages; checking to see if a memory address is out of the bounds of a memory chunk is something that Java restricts at the language level. Graphical debuggers make it easier to see the lower-level information of a program, but do not present the data in its intended context.

Fundamental to typical debuggers is the concept of a breakpoint, a programmer-defined location in the code at which execution becomes suspended. The state of a suspended program can be queried by examining the contents of variables. *Source-level debuggers* make it appear that the user is working with the original source code of the program when in fact they are working with object code underneath. These tools are capable of mapping object code back to the source language level, typically by using external information such as a map from source line locations to instruction addresses, and compiler emitted type information. However, they still require the user to interact at a very detailed level.

Eisenstadt's survey of programmers most difficult bugs [Eis97] looked at methods used to track down errors. He found that there were four major techniques used: gathering data from the program, inspection and hand-simulation of the code, asking other people for help, and observing program behavior under controlled experiments. The most commonly used technique was gathering data, which accounted for more than half (53%) of the bugs.

Gathering Data

Studies have shown that most programmers do not use a debugger to find and fix bugs in their programs [Gra83]. Typically, programmers find bugs by looking for anomalies in the behavior of their program and thinking about a high-level explanation of the cause. Hypotheses are proposed and tested by making changes to the behavior of the program and observing the results. Low-level information, such as the values of variables that are used in intermediate computation, are typically displayed using print statements in the source language.

However, debuggers are often handier than print statements since they do not require the variables to be printed to be known in advance of running the program. The compile-edit-run cycle is avoided with the debugger. They are less intrusive than print statements, as the program does not need to be modified. Despite this, print statements have advantages over debugging. For example, they do not require learning a new tool, and they are programmed in the same language as the source program.

Graphical debuggers offer some usability benefits over print statements in their ability to show state more efficiently than command line tools. For example, the balloon values that appear in modern GUI-oriented debuggers are popular because they allow users to casually pick out variables they are interested in and show their value without having to explicitly type the name of the variable.

Debuggers should go a step further and devise ways in which the large amounts of data can be seen at the higher level in which the programmer thinks about it. Debuggers should be visibility tools; they show the state of a program at a given time.

Other Debugging Issues

There are a number of interesting debugging challenges that this thesis does not address. Compilers that generate optimized code that cannot easily be mapped back into source-level constructs present a major challenge for the implementors of debuggers. See Hennessy [Hen82] for a detailed description. Many modern programs are not single-processor sequential programs, but a collection of separate threads of execution, possibly running on separate machines. A large amount of literature addresses this area of debugging of parallel and distributed programs [BW83, MM91, CLVM98]. A very powerful debugging feature is the ability to inspect the target's previous states. *Reverse execution* allows the use to resume execution at previous points in the program, undoing changes made after that point. See Zelkowitz [Zel73] and Tolmach [Tol92]. Finally, systems that have real-time requirements present additional debugging challenges, since slowing down or interrupting execution of a program may not be possible [TFCB90].

1.2 Debugging Languages

One way to support debugging at a higher level of abstraction is to allow users to use a set of building blocks in the debugger which compose more complex functionality. For instance, the contents of a linked list could be displayed by printing each element of the list individually with the lower level printing commands.

For this reason, several systems support debugging languages. These are typically high-level languages that provide constructs such as loops and procedures for users to create higher level functionality. One of the first debugging languages was Dispel [Joh81], which was used for the RAIDE run-time debugging system [Joh77, Joh78]. A more recent example is the Acid debugger [Win94] for Plan 9.

Debugging languages are useful not only for examining data, but for controlling execution. Procedures can be written that correspond to breakpoints. Such procedures can check state information when the corresponding breakpoint is reached. They can resume execution, print values of target variables, or give control to the user. The Dalek debugger [OCHW91] provides a language that allows the user to define *events* – occurrences of interesting activities in the code execution – as a way to form higher level abstractions during execution.

Debugging languages exist in a variety of forms. Some debugging languages try to match the underlying source language, as in the case of the UPS [Rus92] debugger for C. This approach has the advantage that the user does not have to learn a special purpose language to debug. However, disadvantages are that it is not portable to other target languages, and the language itself may not be effective enough for debugging. High-level languages can be better suited for debugging because they allow fairly powerful actions to be constructed more concisely. For example, the Duel debugging language [GH93] allows the user to find all the elements of an array greater than 1 with a single line query.

The versatility and power that debugging languages provide have a number of useful applications:

- **Automated testing** - Users can write scripts to change values at certain points in the program and control execution. This can be used to execute parts of the code that might not otherwise be covered in a typical run.

- **Code Patching** - Users can attach code to a breakpoint that either replaces part of the original code or fixes it appropriately. This avoids having to modify the original program.

- **Procedure Tracing** - Users can write functions that print out information at procedure entries and exits. These logs can be used to obtain a trace of the program.

- **Profiling** - Procedure tracing can be augmented with time stamps as a means of execution profiling. Also, heap usage can be checked at various points to obtain memory profiling.

The Deet Debugger

This thesis presents a debugger called `deet`, which provides a high-level language for debugging. `deet` is both implemented and programmed with `tksh`, a graphical UNIX shell that provides the graphical facilities of Tcl/Tk [Ous94]. The use of `tksh` has the advantages that it is supports familiar high-level languages (Tcl and shell), and is designed for interactive use. Facilities like conditional breakpoints can be written in either Tcl or the shell.

Let us consider a concrete example of how `deet`'s language could be used for high-level debugging. Suppose we wish to set up a breakpoint that will invoke a user-defined script to check a property of one of the target's data structures. In particular, we want to see if any element of an array called `numArray` is negative whenever execution reaches a particular line in one of our files. With `deet`, we could write the following code segment:

```
function listBreak {
        typeset arr=$1
        integer i s=$(arraySize $arr)
        for (( i=0; i < s; i++ )); do
                if (( $(var "$arr[$i]") < 0 )); then
                        print "Negative element in array"
                        return
                fi
        done
        continue
}

stop at calc.c:37 "listBreak numArray"
```

When the breakpoint is reached, the function `listBreak` is executed; the name of the array (`numArray`) is passed as an argument. `listBreak` checks the elements of the array for any negative values. If no negative values exist, the program continues automatically. Otherwise, a message is printed and the control will be passed back to the user.

Debugger Design

`deet` demonstrates the strengths that languages can have in debugger design. `deet`'s language contains a minimal set of commands that communicate with the target's *nub*, which is a small set of low-level machine-dependent functions. All high-level functionality is implemented on top of these functions. The nub can be embedded in the target program at compile-time, or implemented on top of existing debuggers.

`deet` differs from conventional debuggers in that it is machine-independent, graphical, programmable, distributed, extensible, and small. Whereas most debuggers are large and complicated, `deet` is less than 1,500 lines of shell plus a few hundred lines of machine-specific nub code. It is thus easy to understand, modify, and extend. We present `deet` in more detail in Chapter 2.

1.3 Software Visualization

Another way to debug at a higher level of abstraction is to *visualize* the program. According to the adage, a picture is worth a thousand words. Visualization is defined as *the power or process of forming a mental picture or vision of something not actually present to the sight* [SDBP98]. Although visualization typically involves making pictures, visualization may exist in forms such as sound and smell [DB92]. Visualization is useful for many purposes. One common use is for concisely expressing a large amount of information so that the important features of the information can be understood quickly. For example, the field of scientific visualization deals with taking large sets of numerical data and displaying them (for example with scatter plots) such that patterns in the data can easily be detected.

Software visualization, the application of visualization to the domain of understanding computer software, has long been an important tool for computer programmers. Examples of software visualization are plentiful. Data structures are often diagramed on paper or blackboards as pictures, typically with boxes and arrows. The text of source code is usually formatted so that some structural information about the program can be inferred (for example, indentation of control structures and skipped lines between procedures). Flow charts have been used to explain programs since the 1940's [GvN47].

Software visualization can exist in many forms, and can be broken down into several dimensions. Figure 1.2 shows one taxonomy. Visualizations can exist on the level of the object code, source code, or algorithm. Although sometimes the categories are not easy to distinguish, we typically refer to visualizations at the source code level as *program visualization*, and visualizations at the algorithm level as *algorithm visualization*.

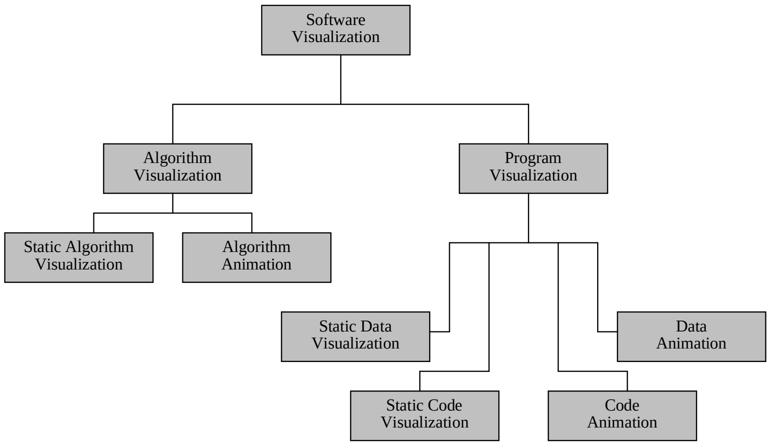


Figure 1.2: Types of Software Visualization

When a program is executed, a sequence of operations occur through time. Visualizations that show how the state of the program changes are referred to as *animated*. Visualizations that depict the state of a system in a single state are called *static*.

Visualizations can be *automatic* or *manual*, depending on the amount of information the observer of the visualization must provide for its construction. Some visualizations can be produced automatically without any additional input from the user. However, most visualizations of this kind fall into the category of program visualization, as semantic information that is needed to produce higher-level visualizations is missing from program code. For example, algorithm animation systems often show two elements of an array being swapped. However, the code for swapping involves copying the first element into a temporary location, copying the second element into the first, and copying the temporary to the second element. It difficult to automatically determine that this sequence of operations should be grouped as swapping the array elements, rather than setting each individual value. Some systems, such as UWPI [HWF90], attempt to infer high-level information automatically from a program and can be categorized as automatic algorithm visualizers.

The two main ingredients of program are its instructions and data. When a program is run, data starts off as input and is manipulated by the code until it arrives at output. Some visualization systems are focused primarily on code; examples of such systems are SEE [Bae88], which visualizes program code by pretty-printing source, and Jinsight [PLVW98], which visualizes execution paths (function calls) from the trace of a dynamically running program. Other systems are focused on data, such as Balsa [BS84], which draws data structures in programs.

The work presented in this thesis addresses the visualization of both code and data in

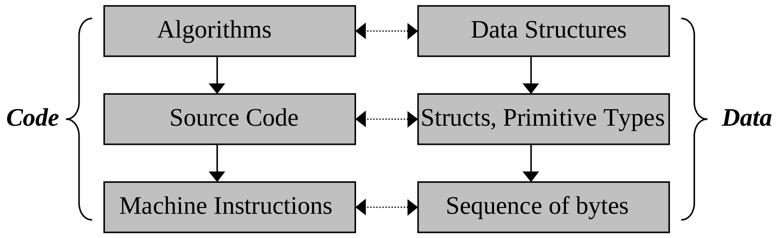


Figure 1.3: Levels of abstraction in software

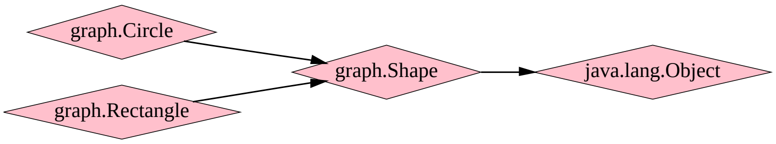


Figure 1.4: Class inheritance diagram

debuggers. Traditional debugging itself can be thought of as a form of software visualization. The debugger is running machine code; the program's source code is actually a visualization of the underlying object code. Further, the printing of values in textual debuggers can be thought of a one dimensional visualization of program data. Figure 1.3 shows the levels of abstraction that can be thought of for debugging.

1.3.1 Visualization of Code

This thesis presents Chava, a system that analyzes and tracks changes in object code, specifically Java class files. The tool extracts information from application or applet code about classes, methods, fields and their relationships into a relational database. Given our data model, a suite of programs that query, visualize, and analyze the structural information were generated automatically from CIAO [CFKW95], a retargetable reverse-engineering system.

Figure 1.4 is the visualization of a class hierarchy computed with Chava. This picture concisely portrays structural information about the program we are debugging. A wide range of queries can be entered to depict other facets of the target program. Chava's tools support a variety of analyses, such as reachability and clustering analysis.

Further, supplementary checksum information in the database can be used to detect changes in two versions of a program. This is useful in debugging, as we can quickly narrow down the possible problem areas in a program given working and non-working versions. Chava can also help spot changes in library components that might impact developers when working with updated versions of third party software, helping to solve some of the hardest bugs involving module interaction.

To adequately work with Java programs in applet form, Chava works with HTML analysis tools to track both the static and dynamic content of web sites. Chava is described in detail in Chapter 3.

1.3.2 Visualization of Data Structures

Much of the early work in the visualization of data structures was inspired by two pieces of work: Sorting out Sorting [Bae81], a movie which visually showed a variety of sorting algorithms in action and compared their performances; and Incense [Mye83], which automatically displayed data structures in Mesa programs. In particular, these pieces of work led to the development of *algorithm animation systems*, which visually animate the execution of a program.

The first interactive algorithm animation system was Balsa [BS84], which aimed to animate any algorithm in a dynamic interactive environment. Balsa worked by allowing programmers to annotate their source code with procedure calls to the animation system. Systems that followed Balsa (Balsa II [Bro88], Zeus [Bro91a], Tango [Sta88], ANIM3D [NB94]) evolved a common structure for animation systems. Systems consist of three parts: the input generator, the algorithm which generates animation events, and the renderer which turns the animation instructions into a graphical view.

These systems work with source code, and require users to annotate code to produce their animations. This is not possible in debuggers, which work at a lower-level. Working with object code presents new challenges for providing software visualization.

In debuggers, source is not always available to us. Old systems are around whose source is long gone or can no longer be compiled. We need to be able to work with such systems. Also, with the growing use of the World Wide Web, we often run applets and other forms of safe object code. If we want to reverse engineer such systems, we need to be able to work without source since source it is not often distributed.

Annotations used in algorithm animation systems are not convenient for debugging. These systems are primarily used when the data to be viewed is known before the program is run. Thus, annotations can be placed into the appropriate parts of the code where the data is constructed. In debuggers, we often don't know what we want to see until during a program run, when the data has already been constructed. Also, programs are frequently debugged after they have already been started, making annotations in previously executed code useless.

The PROVIDE debugger [Moh88] was one of the first debuggers to allow users to manipulate the visual display of data through a user interface. 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. Some modern debuggers have similar features, such as ddd [ZL96] and the SGI Workshop Debugger (Developer Magic Debugger) [SGI95].

Recent systems have attempted a more flexible and generalizable approach. The Lens debugger [MS94] applies the techniques found in algorithm animation systems to the debugger level. A system called Pavane [RCWP92] allows users to define predicates that map data structures to visually displayed objects. These systems are described in more detail in Chapter 4.

Example: Sorting

To illustrate the benefits of software visualization in debuggers, let us consider a concrete example. A common operation in programs is sorting an array of elements. Sometimes we work with a single array of elements that we wish to sort in a variety of ways. Rather than maintain separate copies of the array for each sort, a common technique is to create an array of indices that refer to the array being sorted, and then sorted the array of indices. This is known as *indirect sorting*.

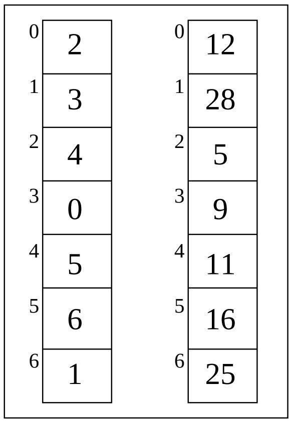


Figure 1.5: Indirect Sorting: Program Visualization

Suppose we are indirect sorting an array of integers. There are two arrays we are working with, the array of indices and the array of values. These arrays are related to each other in that the values of the index array refer to indices in the value array. If we used a debugger to automatically print out the arrays, we would probably see something close to what is drawn in Figure 1.5. This does not display much useful information, as we would like to see where the values are now positioned using the indices in the indirect array. An observer has to look at the two arrays, examine the values in the indirect array, and figure out what the target array should look like. The debugger has no way of knowing how to relate the two arrays automatically, since this is higher-level information about the algorithm that is not conveyed at the programming language level.

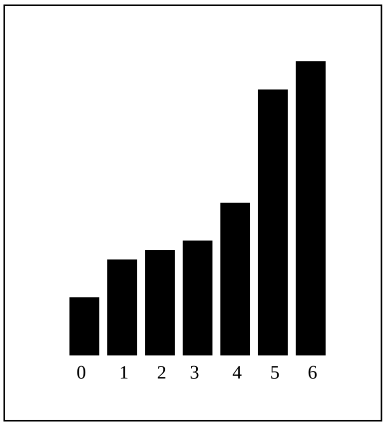


Figure 1.6: Indirect Sorting: Algorithm Visualization

If we could allow the user to specify how the arrays are related, we could end up with a picture like the one shown in Figure 1.6. This thesis presents a system called TraVis that supports such specification.

Data Visualization in Debuggers

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 thesis 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. Traversal-based visualization and TraVis are discussed in Chapter 4.

1.4 Contributions

The systems presented in this thesis address a variety of problems that arise in the area of debugging at a higher level of abstraction. Deet, presented in Chapter 2, offers an improved design for graphical debuggers and supports a high-level language for abstract debugging. Chava, presented in Chapter 3, uses program visualization to show relationships between various entities of the target program. TraVis, presented in Chapter 4, supports algorithm visualization in debuggers using a new method called traversal-based visualization which is used to gather data. The major contributions made by this thesis are:

- **Use of familiar graphical language for debugging.** Debuggers that support user scripting face the problem of deciding which language should be used. Programming the debugger with the same language of the debugged target is prone to portability problems and is too low-level. High-level debugging languages suffer from being debugger specific and therefore difficult to learn. Deet addresses this problem by presenting `tksh`, a language that interprets two general purpose high-level languages, shell and Tcl. The interactive features of shells make `tksh` a good fit for debugging.

- **Piece-parts design for debuggers.** It is important to design software with the right abstractions. An important lesson learned from this research is that choosing good abstraction promotes software reuse. In particular, `deet`'s design results in a powerful debugger that is an order of magnitude smaller than traditional debuggers such as `gdb`. The use of the debugging nub allows us to mix and match debugger front-ends and back-ends. We can use `deet`'s interface with a variety of languages such as C and Java by using alternate nub implementations. Our nub can be emitted by the compiler (resulting in a fully machine independent debugger), or as a layer on top of existing debuggers such as `gdb`. We can switch the front-end from a graphical user interface to a textual interface emulating `gdb`.

- **Application of data repositories to object code.** Reverse-engineering systems that visualize software are popular tools for programmers. However, they require access to source code and often use too much space and take to much time to be practical for use in debugging systems. Chava applies existing reverse-engineering tools to a repository built entirely from object code. The performance analysis of Chava shows that not only can such repositories can be built without source, they can built efficiently. In fact, generating a repository from a Java object file with Chava takes about the same time as parsing its source.

- **The method of traversal-based visualization and its evaluation.** This thesis presents a new technique for abstractly displaying data structures in a target program. Unlike the technique of *interesting events* [BS84], it does not require annotation of source code. Instead, it declares a mapping from the low-level data to high-level entities that will be displayed. This is well suited for debugging, since we work with already running programs and may not know the data we wish to see in advance.

  In addition to presenting this new method, we evaluate its usefulness by studying user response to a functioning system, TraVis, which uses traversal-based visualization. Results from usability experiments presented in Chapter 6 show encouraging response to the idea.

  Traversal-based visualization also provides ways for the abstract displays to be modified, allowing the changes to be reflected in the underlying data. Most debuggers that support high-level display of data do not have support for this.

- **Use of database visualization for algorithm visualization.** The design of TraVis separates the gathering of data to be visualized from the display of the data. Instead of directly drawing entities produced from the traversal of the data, a database is created which holds information about each on-screen component to be shown along with its attributes. A separate module of TraVis converts the database into a graphical representation.

  There are numerous advantages to this novel approach. The same information can be visualized in different ways by using other layout modules with the same database. Alternatively, the same layout modules can visualize data from alternate sources and systems as long as the data conforms to the format of the database. Further, we can leverage existing database technology, such as database visualization and differencing tools, with our data.

After presenting each of the systems in the next few chapters, we discuss conclusions and ideas for future work in Chapter 7. Portions of the work described in this thesis have been previously published as references [HK97], [KA98], and [KCK99].

Chapter 2

Deet

2.1 Introduction

Traditional debuggers are indispensable tools for locating and fixing program errors. Despite their importance and pervasiveness, they continue to harbor inadequacies that limit their usability. For example, UNIX debuggers typically have textual user interfaces that are cryptic at best. When debuggers are hard to use, programmers tend to litter their programs with print statements instead of using a debugger.

Debuggers, like compilers, must deal with portability issues. Debuggers are notoriously machine-dependent programs; they depend on the target architecture, operating system, compiler, and linker. Thus, porting a debugger from one system to another can require a substantial amount of effort. For example, about one-third of `gdb`'s source code is machine-dependent.

Few debuggers have programming facilities in which programmers can write application-specific debugging code. Such code is useful for nontrivial queries of data structures, such as displaying the second to last element in a linked list, or all the positive elements in an array. Other examples include setting conditional breakpoints and automating program testing. Debuggers that support programming facilities do exist, but often the language is idiosyncratic either to the debugger or to the source language, or both, and hard to learn.

Most debuggers are large and complex programs; for example, `gdb` [SP91] is about 150,000 lines of C. This complexity has some unfortunate consequences. First, debuggers are often themselves buggy, because, like any large program, their complexity and size makes them prone to errors and to inconsistent behaviors on different platforms. Second, debuggers are usually difficult to extend, because their implementations may be hard to understand and to modify.

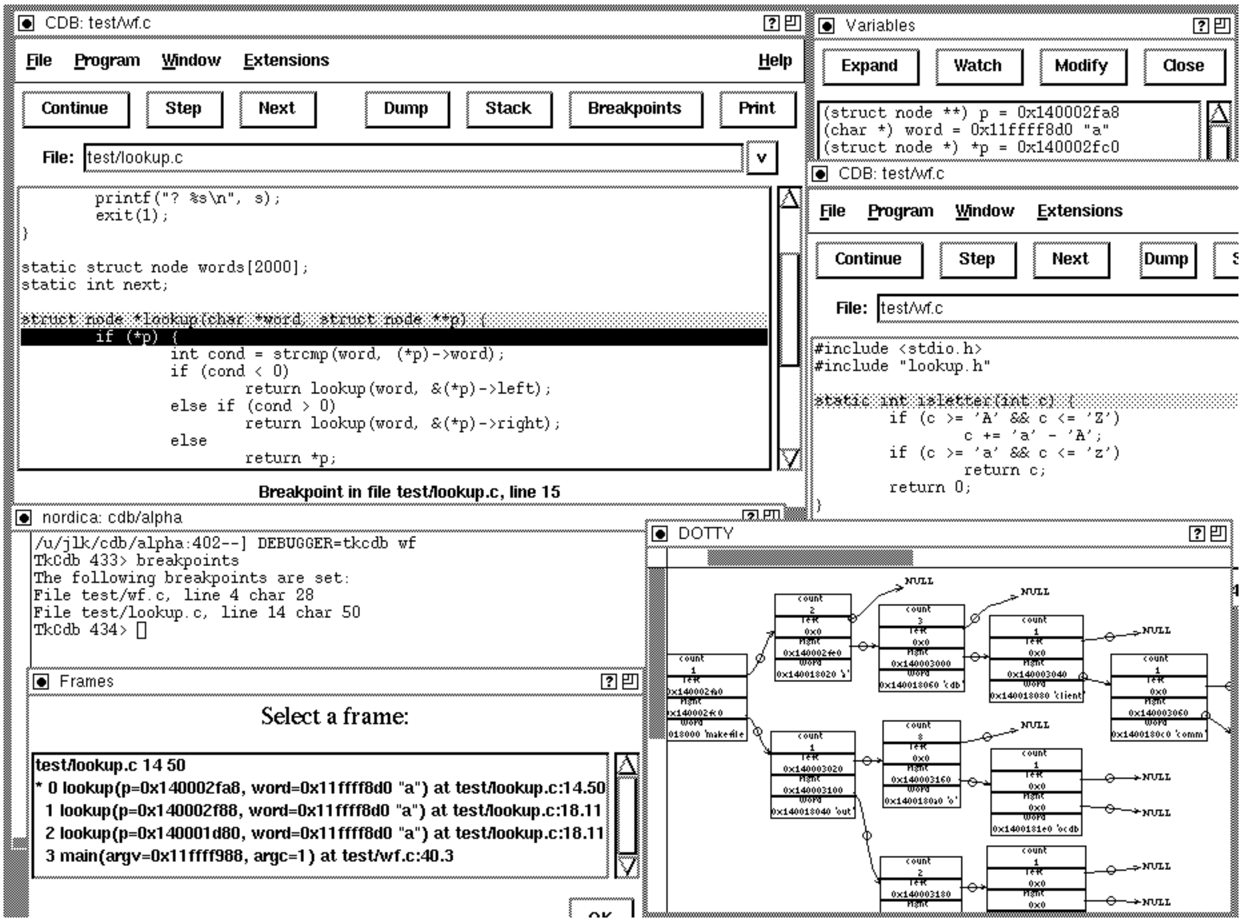


Figure 2.1: `deet` screen dump

`deet` (*d*esktop *e*rror *e*limination *t*ool) addresses these shortcomings. It provides both a textual and graphical interface to make it easy to use. Users can perform most debugging actions by pointing and clicking, and data structures can be displayed graphically. The GUI is written with Tk [Ous94]. `deet` is also programmable: Its capabilities can be extended by writing in either Tcl or the shell.

Nearly all of `deet`'s implementation is machine-independent. It uses a small "nub" which provides facilities for communicating with the debugger and controlling the target. The nub-based approach permits `deet` to debug a target running on another machine. Figure 2.1 shows the screen of a typical debugging session. `deet` doesn't attempt to match debuggers like `gdb` feature-for-feature; for example, `deet` can't examine core dumps, evaluate arbitrary C expressions, or debug at the assembly-language level. Nevertheless, its implementation is surprisingly simple. Its complete source is approximately 2,500 lines of shell and C.

`deet` is based on `cdb` [HR96]. `cdb` is a machine-independent debugger that eliminates ma-

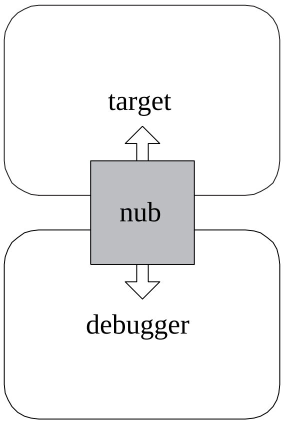


Figure 2.2: `cdb`'s design

chine dependencies by adding a small amount of information into the target program at compile time. `cdb` communicates with the target through a nub—a small machine-independent interface that constitutes the core functionality of the debugger, as suggested by Figure 2.2. The nub implementation can be made machine-independent, as `cdb` shows, but machine-dependent implementations are also possible and are undoubtedly more efficient. The nub is small enough that reimplementing it for new platforms is nearly as easy as porting the machine-independent implementation.

There are four components of `cdb`:

1. The nub interface, which stands between the debugger and machine-dependent target manipulations.

2. The nub implementation, which consists of the nub interface functions, special code emitted by the compiler to support the nub, and a wrapper around the linker to load the nub and the machine-independent symbol table.

3. A machine-independent symbol-table format, which is emitted by the compiler and linked into the target.

4. A simple, text-based debugger that uses the nub to provide minimal functionality; this debugger is intended to be replaced with more sophisticated debuggers, like `deet`.

Any of the last three components can be replaced with alternative implementations. For instance, the nub can be replaced with a machine-dependent implementation that uses the `ptrace` system call like most UNIX-specific debuggers, or a layer on top of `gdb`. The machine-independent symbol tables could be replaced with the usual machine-dependent "stab" symbol tables embedded in UNIX executables. Finally, the debugger itself could be replaced with any program that uses the nub interface. `deet` is a replacement of this fourth component. Using `deet` does not directly involve changes to any of the other components, but implementing `deet` did induce additions to the nub and to the symbol-table format beyond their original designs.

`deet` is written in `tksh` [Kor96], a version of the new Korn shell [BK95, KNK96] that has been extended to support Tcl. This includes a C library that can be used to manipulate the state of the Tcl interpreter, such as reading and writing variables and creating new built-in commands. `tksh` can run any library written on top of the Tcl library, which includes the Tk graphics library. Thus, Tk commands, like `button` and `pack`, can be invoked from `tksh` scripts.

`tksh` should be thought of as an extension to Tcl rather than as an alternative to it. `tksh` allows Tcl scripts to be run directly with the `source` command. Tcl scripts share variables and functions with `tksh`, allowing Tcl scripts to work with shell scripts.

`tksh` is used as the debugging language for `deet` primarily because of its strengths as an interactive command language. Debuggers are interactive programs. `deet` takes advantage of the interactive facilities of `tksh`, such as command-line editing, job control and pipelines. Using the command-line interface to `deet` feels like using a shell because the debugger itself is an extension of `ksh`. `tksh` also offers familiar, high-level languages. Many programmers already know how to write shell and Tcl scripts, which is 90% of what's needed to use `deet`. However, a `perl` or `python` programmer could rewrite the `deet` front end and still use any existing nub implementation.

`deet` itself is written in `tksh` on top of a set of additional built-in commands for debugging. This code is simpler to understand than the corresponding C code would be, because it's written in a high-level language. `deet` can also can be modified during a debugging session to suit specific applications.

2.2 Using `deet`

`deet`'s features are best explained by seeing it in action. First, the target program is compiled with `lcc` [FH95] with the appropriate debugging option to embed the nub into the target:

```
$ lcc -Wf-g4 wf.c lookup.c
```

Here and in the displays below, *`slanted`* type identifies user input. When the generated `a.out` is executed, the debugger specified by the environment variable `DEBUGGER` is also started, so

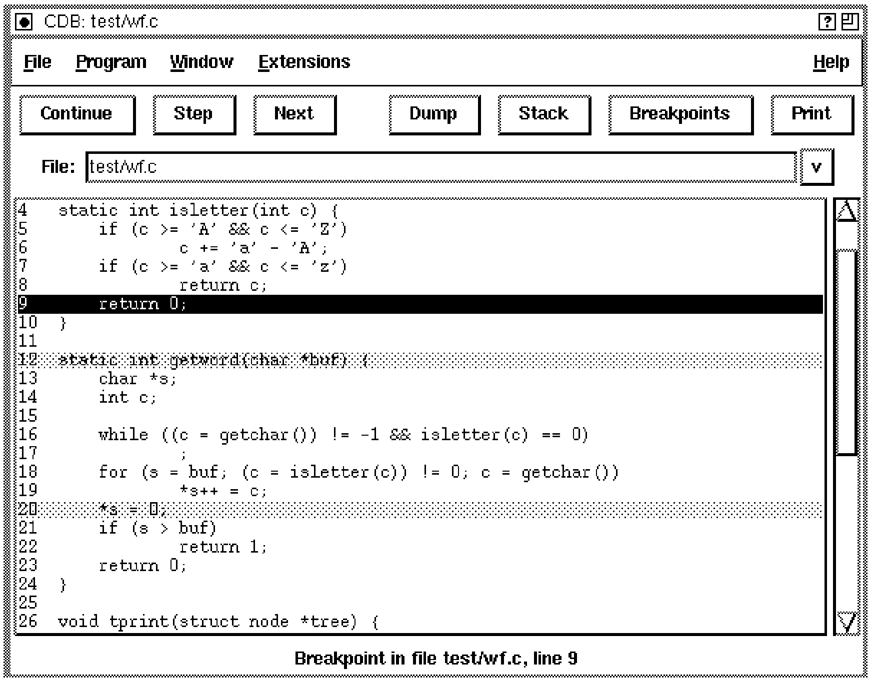


Figure 2.3: `deet`'s source window

```
$ DEBUGGER=deet a.out
```

starts both `a.out` and `deet`. At this point, the source window shown in Figure 2.3 appears. The user is prompted for textual `deet` commands in the shell window from which `a.out` was invoked, but most debugging actions are performed with the mouse.

Single-clicking on a line highlights that line if it contains a breakpoint; double-clicking on the line sets the breakpoint. `deet` can set breakpoints on expressions, not just statements, so there may be more than one breakpoint in a line. When a line has multiple breakpoints, double-clicking sets the breakpoint closest to the cursor. Breakpoints are indicated in the window in grayed reverse video. Double-clicking on a breakpoint that has already been set removes the breakpoint.

The breakpoints window, like the one shown in Figure 2.4, displays a list of all breakpoints and related information about each breakpoint, such as its location and break condition. These conditions are `deet` expressions that are evaluated whenever the breakpoint is reached; if the condition is true, the target stops. When the target stops at a breakpoint, the current source window shows the file and line number of the breakpoint, and reverse video highlights the line containing the breakpoint. A condition can be changed by highlighting the breakpoint in the breakpoints window and editing the condition field, and a breakpoint can be removed by clicking the "Delete" button in this window.

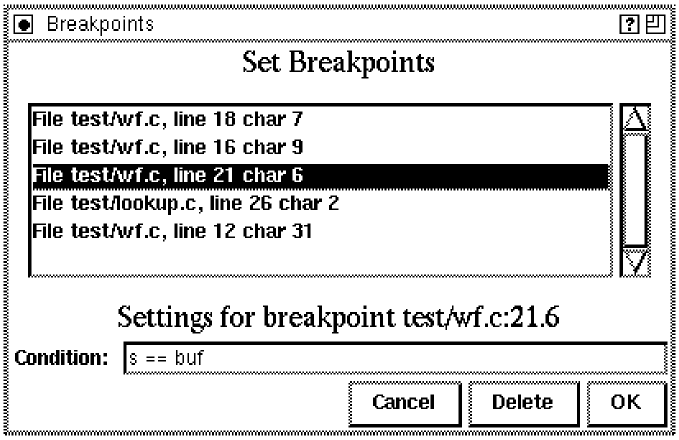


Figure 2.4: Breakpoints window

The stack can be shown by clicking on the “Stack” button in the source window (see Figure 2.3). This displays a new window that shows each frame on the stack, from the top down, as illustrated in the lower left corner of Figure 2.1. An individual frame can be selected, and clicking a button in the stack window performs the corresponding action on that frame. For instance, by clicking the “Dump” button, the names and values of the parameters and locals for that frame are displayed. Clicking the “OK” button causes the source window to display the file and line number of the call to the selected frame.

Highlighting a variable in the source window and clicking “Print” causes a pop-up window to display the value of that variable (see the upper right corner of Figure 2.3). If the variable is a pointer, a structure, or a union, double-clicking on the variable expands its value. For pointers, the value of the referent is displayed; for structures and unions, the values of the fields are displayed. `deet` will also display the value of a variable in a balloon help style popup window if the cursor is left on top of it for sufficient time, similar to Microsoft's Visual C++ debugger.

A variable can be modified by clicking “Modify” in the variable window, which prompts the user to enter a new value. A variable may also be watched, which causes its value to be displayed in the variable window and updated as execution passes each potential breakpoint.

Commands may be typed at the debugger in the shell window in a manner similar to `gdb`. For instance, the `breakpoints` command displays the current breakpoints:

```
deet> breakpoints
The following breakpoints are set:
File test/wf.c, line 4 char 28
File test/lookup.c, line 14 char 50
```

Commands are just `tksh` commands, so shell commands like `history`, `pwd` and `make` can be entered as well.

Most of the state in a `deet` debugging session can be saved and restored later in a subsequent, separate debugging session. This state includes breakpoints and their conditions, locations of files, and user-defined `tksh` functions. `deet` saves the state by writing a shell script that can be interpreted to restore the state.

2.3 The Debugging Nub

The nub interface is designed to be as small as possible while supporting the fundamental debugging operations common to all debuggers. The complete API is as follows.

`_Nub_init`	initialize the nub
`_Nub_set`	set a breakpoint
`_Nub_remove`	remove a breakpoint
`_Nub_src`	visit breakpoints with a given pattern
`_Nub_frame`	move to a specific stack frame
`_Nub_fetch`	read the target's memory
`_Nub_store`	write the target's memory

The nub does not support high-level facilities, such as expression evaluation or specific symbol-table formats, because these facilities can be implemented by other interfaces or by debuggers themselves.

`_Nub_set` and `_Nub_remove` set and remove breakpoints, which are specified by a file name, line number, and character position. Unlike most debuggers, `deet` specifies breakpoints by the locations of expressions, not lines. So, for example, it is possible to set a breakpoint on the increment part of a C `for` loop.

`_Nub_src` accepts incomplete breakpoints, in which any of the file name, line number, or character position are omitted, and invokes a debugger callback on all possible breakpoints that "match" the incomplete one. `deet` uses this function to determine which breakpoint to set when a user clicks on a line.

`_Nub_fetch` and `_Nub_store` access the target's memory. They accept a buffer address, a byte count, and an address space identifier, and read/write data from/to the target. The address space identifier may specify an operating-system address space, such as the text or code segments. It can also specify logical address spaces that may not be part of the target, like the symbol table, for example. It is the nub's responsibility to access the appropriate data. Debuggers can view all data about the target as if they were stored in memory.

`lcc` emits machine-independent symbol-table data into the target's address space, and `_Nub_fetch` reads these data. Another `cdb`-specific, but machine-independent, interface provides a higher-level view of the C symbol table as an inverted tree of symbol objects. If we were using the nub with, say, Modula-3, this high-level interface would have to be replaced with one specific to Modula-3, and that interface would use `_Nub_fetch` to read the symbol table generated by the Modula-3 compiler. There's nothing special about `lcc`; other C compilers could be used given an appropriate nub implementation. It's the nub that's the critical component, not the compiler.

`deet` includes versions of the nub and symbol-table functions for use with Tcl or `tksh`. These `tksh` commands differ from the C routines in two ways: They are at a higher level, because they manipulate source-level symbols, types, and values, and they accept and return strings, so that they can be used in Tcl or `tksh` scripts. The complete list is:

`db_open`	initialize the target
`db_breakpoint` { `-set` | `-delete` | `-list` } *file line character*	
	set, remove, and list breakpoints
`db_frame` [*n*]	get/set current frame
`db_getval` *type address*	read a value of *type* from *address*
`db_putval` *type address value*	write the *value* of *type* to *address*
`db_continue`	resume execution
`db_sym` { `-all` | `-files` | `-locals` | `-params` | `-name` *name* }	
	finds the symbol-table entries
`db_type` *type*	get *symbol*'s type information

`db_breakpoint` is a combination of `_Nub_set`, `_Nub_remove`, and `_Nub_src`. The `-set` option sets all of the given breakpoints, which might be incomplete; that is, *file* can be `""`, and *line* and *character* can be zero. The `-delete` option removes breakpoints, and the `-list` option lists possible breakpoints.

`db_frame` is equivalent to `_Nub_frame`: With no arguments, it returns the current frame as a Tcl list containing the frame number, the function name, and a file, line number, character number triple that gives location of execution within that frame. With an integer argument n, `db_frame` makes frame n the current frame and returns the null string. Frames are numbered from the top of the stack, beginning with zero.

`db_getval` and `db_putval` commands are similar to `_Nub_fetch` and `_Nub_store`, but require type information to be specified along with the value, because Tcl deals only with strings. Tcl cannot, for example, deal directly with binary floating-point values or with structures. Types are specified by type identifiers, which are just generated strings. `db_getval` returns a string representation for the value of *type* at *address*, and `db_putval` writes the *value* of *type* to locations beginning at *address*.

`db_sym` and `db_type` return symbol-table data. A symbol-table entry is a Tcl list { *name*, *type*, *address* }. `db_sym`'s `-all` option returns a list of all of the symbols in the target; that is, a list of three-element lists. The `-files` option returns a list of all of the source files in the target. The `-locals` and `-params` options return lists of the locals and parameters for the current frame. The `-name` *name* returns the symbol-table entry for *name*, or an error if *name* is not a visible symbol.

`db_type` returns a string describing the type represented by identifier *type*. If *type* represents int, `db_type` returns `"int"`. Similarly, if *type* represents T `*`, E `[`n`]`, or a structure type, `db_type` returns, respectively, the type identifier for T, n and the type identifier for E, and a list of names and type identifiers for the fields.

2.4 Programming with `deet`

Much of `deet` itself is written in Tcl and `tksh`, using the `db_*` nub commands described above. Users can extend `deet` by writing Tcl and `tksh` commands; for example, features like conditional breakpoints and nontrivial program queries can be written in `tksh`. `deet` can also be extended by external programs. This section illustrates some typical extensions.

Simple extensions can be written directly in `tksh`. Appendix B gives an overview of the `tksh` language. For example, the following script displays all of the null elements in an array, the name of which is supplied as an argument.

```
function nullElements {
        typeset arr=$1
        integer i s=$(arraySize $arr)
        for (( i=0 ; i < s ; i++ )); do
                if [[ $(var "$arr[$i]") == 0x0 ]]; then
                        print "Element $arr[$i] null"
                fi
        done
}
```

`nullElements` uses two external `tksh` functions: `arraySize`, which returns the number of elements in an array, and `var`, which returns the value of a variable. These functions are provided as part of `deet`. The for loop visits each element of the array specified by the first argument, retrieves its value, and prints the array name and index of the null elements.

User-defined functions can also manipulate `deet`'s interface. For example, if we're checking repeatedly for null elements in `hashtable`, we can construct a button to do the job in one click:

```
toplevel .null
pack $(button .null.b \
        -text "Print Null Elements" \
        -command "nullElements hashtable")
```

This code builds the button:

Tcl scripts can be invoked with the `source` command, which is a `tksh` built-in. `source` uses the Tcl parser to parse its input, and uses `tksh` variables and functions in variable and command substitutions. Here's a simple example:

```
function foo {
        X=37
        print "$(bar test)"
}

source <<'EOT'
proc bar {args} {
        global X
        set X [expr $X + 1]
        return "Proc bar: args: $args, X: $X"
}
EOT
```

A call to `foo` prints

```
Proc bar: args: test, X: 38
```

Note that the Tcl procedure `bar` can use and modify the shell variable `X`. Tcl source code can also invoke `tksh` functions and built-ins.

`deet`'s name space is separate from the target's name space. Accessing a target variable from a `tksh` script requires a special function, `var`, which uses the target's symbol table to lookup the variable name and retrieve its value. `var` is sufficient for one-shot lookups, but it's tedious for repeated uses of specific target variables. For these uses, `deet` provides `linkvar` *name*, which creates a new shell variable that is essentially an alias for the target variable *name*. `linkvar` is implemented with *discipline functions*, which are similar to trapped variables in SNOBOL4 [Han76]. A discipline function is a shell function that is associated with a variable, and that function is invoked whenever the variable is read or written. Thus, associating the function

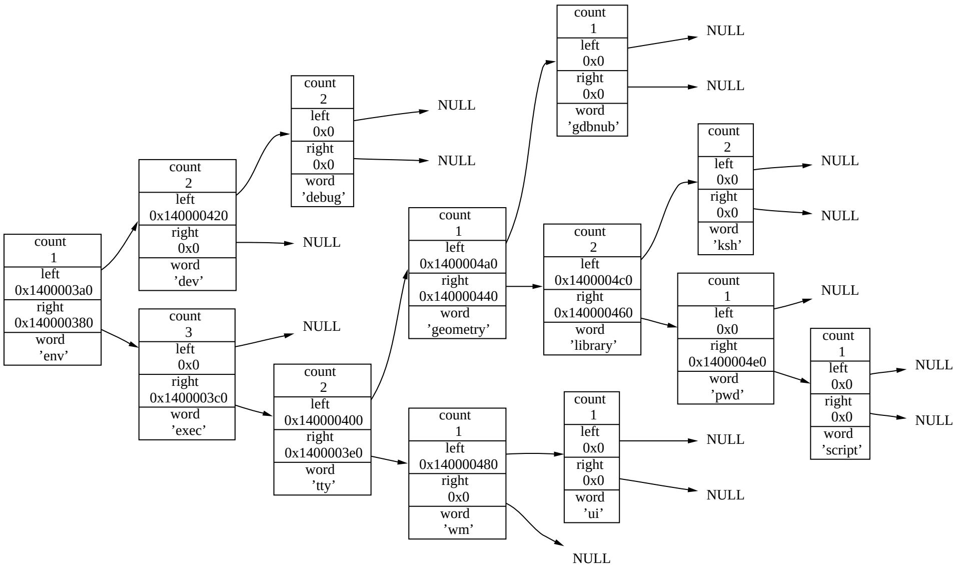


Figure 2.5: Tree generated with `deet` and `dotty`

```
function foo.get {
        foo="$(var foo)"
}
```

with `foo` arranges for the target variable to be fetched every time the shell variable `foo` is read.

`deet`’s capabilities are easily extended by writing Tcl and `tksh` scripts that use the built-in debugger commands. An important advantage using a shell as the debugging language is that the shell can use *any* external tool. For example, it’s relatively easy to extend `deet` to display linked data structures graphically as directed graphs. This feature is similar to that provided by the Data Display Debugger (`ddd`) [ZL96], but the implementation is much simpler, because `deet` uses existing tools instead of building its own facilities. `deet` runs `dotty`, a program for drawing directed graphs [NK94], to draw the graph, sending it the appropriate input for the data structure of interest. Figure 2.5 shows an example of `dotty`’s output. The `tksh` script that invokes `dotty` is only about 60 lines of code, and it handles any linked data structure.

2.5 Implementation

`deet` is written in `tksh`; each `deet` command is implemented as one or more `tksh` functions that call the built-in Tcl nub commands. An example is the `b` function shown in Figure 2.6,

```
function b {    # breakpoint [action]
        integer char=0 line=0
        typeset file point="$1" action="${2-':'}" msg
        eval set -- $(cvtbp "$point")
        file="$1" line="$2" char="$3"
        typeset bp="$(db_breakpoint -list "$file" "$line" "$char")"
        eval set -- $bp
        if (( $# > 1 )); then
                msg="Pick one of $bp"
        elif (( $# < 1 )); then
                msg="No breakpoint in on line $line char $char"
        else
                db_breakpoint -set "$file" "$line" "$char"
                set -- $1
                breakpoint["$1:$2.$3"]="$action"
                [[ $CdbWindow ]] && TextDispBpOn $2
        fi
        [[ $msg ]] && print -- "$msg"
}
```

Figure 2.6: Implementation of the `b` command

which uses `db_breakpoint` to set breakpoints. The size of this function is as important as its details: most debugging features are easily implemented in tens of lines of `tksh` code.

`b` begins by converting its first argument into file, line number, and character number values. When an incomplete breakpoint is specified, some of these values will be converted to null values. For example, `b 8` causes the `cvtbp` function for the argument `8` to become the value of `line` and for `file` and `char` to be null. Next, `b` invokes `db_breakpoint -list` to list all breakpoints matching the incomplete breakpoint. If there is more than one match, a list of possible breakpoints is displayed and no breakpoints are set. If there are no matches, a diagnostic is issued. Finally, if there is exactly one match, that breakpoint is set. The associative array `breakpoint` keeps track of the set breakpoints. The nub doesn't keep track of breakpoints because it is designed to do as little as possible. If the second argument specifies a condition for the breakpoint, it's stored as the value for the `breakpoint` array entry. Finally, the source window (if it exists) is updated to highlight the set breakpoint.

The nub interface can set and remove breakpoints, but it cannot single-step the target [HR96]. `deet`'s `step` function implements single-stepping by setting and removing breakpoints:

```
function step {
        if [[ $deetMode != "step" ]]; then
                db_breakpoint -set "" 0 0
```

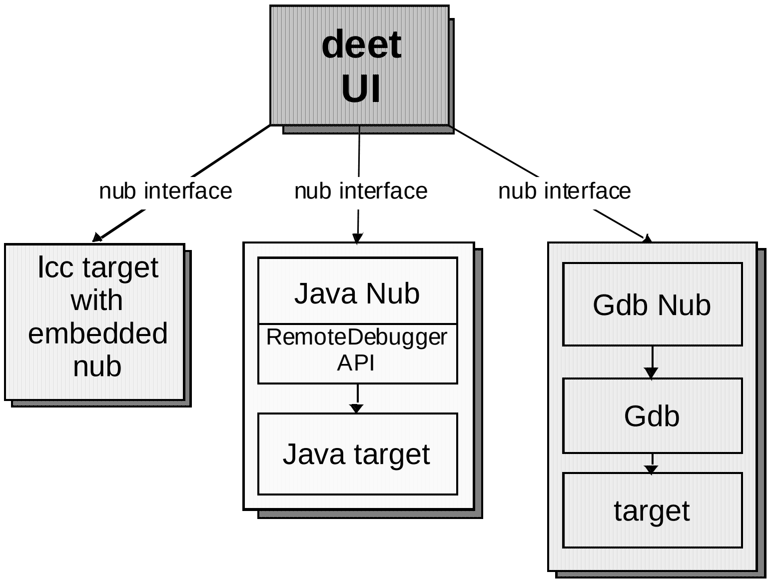


Figure 2.7: Piece-parts design of `deet`

```
        fi
        deetMode=step
        deet_go         # resume execution
}
```

Calling `db_breakpoint` with null values for the file, line number, and character number sets every breakpoint. Implemented naively, setting every breakpoint is expensive in large programs. But, the nub could recognize this special case and use a more efficient implementation. As described in Section 2.7, our implementation of the Tcl nub functions on top of `gdb` exploits this possibility.

2.6 A Nub for Java

An important aspect of `deet`'s "piece-parts" (see Figure 2.7) design is that superior replacements could be used for each part without disturbing the others. For example, a more efficient, machine-specific nub could be used in place of `cdb`'s machine-independent nub; or a better or more familiar user interface could be used.

To demonstrate this flexibility, we've implemented three alternative versions to `deet`'s pieces. First, we've implemented a version of the nub that works on top of Java instead of C. Second, we've implemented a version of the nub which works by communicating with the `gdb` debugger. Finally, we've implemented a replacement for the user interface component that emulates `gdb`'s command line interface. The Java nub is described in this section, the others are described in the next.

The Java Developer's Kit contains a debugging package (set of classes) which can be used to explore and control the state of a target Java program. These classes exist so that a variety of Java debuggers can be implemented using these classes. Java comes with a simple command line debugger called `jdb` which is implemented using the debugging package, but the debugging package is intended to be reused so that more sophisticated graphical debuggers can be written. The package works by spawning off a version of the Java runtime with the target and communicating with it via message passing.

Although it may be possible for the Java nub to directly spawn the runtime and target, we have implemented the nub using the debugging package for reasons of simplicity. Implementing a nub for Java was only a matter of writing the nub interface on top of Java's debugging methods. The code is merely a couple of hundred lines of Java. The routines in the nub work by reading messages off of a socket, processing them using the Java debugger package, and writing the result message back to the socket. A central routine reads messages, decodes them, and calls an appropriate method to handle the corresponding message.

Figure 2.8 shows a method called `frameCmd`, which performs the same functionality as `db_frame`. When called with an argument, it sets the current frame to the number specified with the argument. This is accomplished by finding out the current frame with the `getCurrentFrameIndex` method and using the `up` and `down` methods appropriately. If no argument is supplied, `frameCmd` returns information about the current frame. The needed information is returned using methods in the `RemoteThread`, `RemoteStackFrame` and `RemoteClass` classes.

Although the implementation of the Java nub requires little effort, the result is powerful. With a couple hundred lines of additional code, `deet` can be used to debug Java programs with the same set of features that are used to debug C. Unfortunately, the nub interface does not currently support threads, which limits the usefulness of the Java debugger. However, it would likely be a similar exercise to write a nub on top of Microsoft's debugging API which is part of Windows. Any system that has a debugging interface is easy to port since it is only a matter of implementing the nub interface on top of another debugging interface.

2.7 Comparison with Gdb

As mentioned in the previous section, we've used `gdb` to build a variant of `deet` that uses `gdb` as the nub, and a variant that uses `gdb` as the user interface. These limited experiments

```
public void frameCmd(RemoteThread t, String args[]) throws Exception {
        if (args.length == 2) {
                int oldFrame = t.getCurrentFrameIndex();
                int newFrame = Integer.parseInt(args[1]);
                try {
                        if (oldFrame < newFrame)
                                t.up(newFrame-oldFrame);
                        else if (oldFrame > newFrame)
                                t.down(oldFrame-newFrame);
                } catch (ArrayIndexOutOfBoundsException e) {
                        outputError();
                }
        } else {
                RemoteStackFrame s = t.getCurrentFrame();
                RemoteClass c = s.getRemoteClass();
                outputItem(t.getCurrentFrameIndex());
                outputItem(c.getName() + "." + s.getMethodName());
                outputItem(c.getSourceFileName());
                outputItem(s.getLineNumber());
                outputItem("0");        /* No support for char position */
        }
}
```

Figure 2.8: Implementation of the `db_frame` command for Java

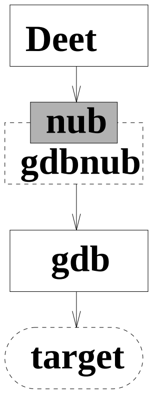


Figure 2.9: Emulating the nub with `gdb`

also reveal strengths and weaknesses in the nub-based design. If `gdb` cannot emulate the nub, then a simple nub offers facilities beyond those of some popular debuggers. If the nub cannot support `gdb`, then the nub is missing some important facilities.

Figure 2.9 shows how `gdb` replaces the nub. `gdbnub` communicates with `gdb`, which runs as a separate process. `gdbnub` translates nub function calls into `gdb` commands, sends these commands to `gdb`, and parses `gdb`'s responses. Another approach would have been to modify `gdb`'s code, but past experience shows that modifying `gdb` is a painful process [GH93]. Using `tksh` makes the approach illustrated in Figure 2.9 *much* simpler: the implementation takes only about 500 lines.

The only nub feature that was not possible to implement with `gdb` was setting breakpoints on any expression. Some nub functions were relatively easy to implement, but not efficiently. For example, `gdb` doesn't provide a way to list all of the possible breakpoints in a file. This was implemented by attempting to set a breakpoint at every line in each file, and checking which breakpoints were successfully set. Fortunately, listing all of the possible breakpoints is rarely done; listing the breakpoints in a specific line is the common usage.

As described in the previous section, `deet` steps through a program by issuing the nub command to set every breakpoint in the target, which is very inefficient. `gdbnub` takes advantage of `gdb`'s single-stepping feature by using the `db_breakpoint` implementation shown in Figure 2.10. When null values are passed to the `-set` option, a variable is set that puts the `gdbnub` into "stepping" mode when execution is resumed. Similarly, when null values are supplied with the `-delete` option, `gdbnub` reverts to "continuation" mode, and all temporary breakpoints are removed.

When `gdb` is used as the nub, `deet` can be thought of as a graphical front end to `gdb`. It provides facilities similar to `ddd`'s, which is also a front end for `gdb`. However, `ddd` is not

```
function db_breakpoint {
        typeset i action=list
        case $1 in
        -l*)    shift ;;
        -s*)    action=set ; shift
                if [[ $1 = "" && $2 = "0" && $3 = "0" ]]; then
                        sendCommand "b main"
                        CONT_CMD="step"
                        return 0
                fi ;;
        -d*)    action=delete ; shift
                if [[ $1 = "" && $2 = "0" && $3 = "0" ]]; then
                        sendCommand "delete"
                        for i in "${!GdbBreakpoint[@]}"
                        do unset GdbBreakpoint[$i]
                        done
                        CONT_CMD="cont"
                        return 0
                fi ;;
        esac
        ...
}
```

Figure 2.10: Implementing single-stepping

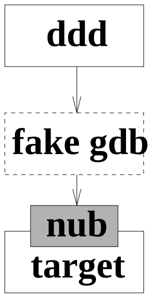


Figure 2.11: Running gdb with the nub

programmable.

Implementing gdb on top of the nub is difficult, because gdb is a huge program and has a large number of features. Some of the features in gdb are inherently absent in the nub. For instance, gdb allows the target to be examined at the machine level; gdb can examine registers and single-step instructions. The nub interface is machine-independent, so it cannot provide these machine-dependent features. Similar caveats apply to data watchpoints, which gdb supports on machines with the appropriate hardware. gdb also supports features that are irrelevant to debugging, per se, such as controlling terminal modes and displaying gdb-specific online help.

Our second experiment thus focuses on only those gdb features used by a graphical front end, like ddd. Figure 2.11 shows the organization of this experiment.

The implementation of gdb using the nub was written with ksh commands. gdb's frame command illustrates the general approach. frame controls the current frame in the stack of the target. Invoking frame with an argument directs gdb's attention to a specific frame. If no argument is specified, the current function with its arguments and location is displayed. For example:

```
(gdb) frame
#0  lookup (word=0x11ffff8e0 "a", p=0x140000010) at test/lookup.c:15
```

Figure 2.12 shows the tksh implementation of frame. The function uses db_frame to move the nub's attention to a new frame. It also uses db_sym and db_getval to fetch and display the frame's parameters and their values.

This implementation of gdb, although incomplete, is only around 1,000 lines of ksh. It implements enough features to support ddd. gdb features that were not implemented include:

- Debugging a target that is already running, which gdb can do on machines where this is possible.

```
function frame {        # [num]
        [[ $1 != "" ]] && db_frame $1 2> /dev/null
        set -- $(db_frame)
        typeset num=$1 name=$2 file=$3 line=$4 char=$5
        typeset params="$(db_sym -params)" p result
        result="#$num  $name("
        eval set -A parm $params
        for p in "${parm[@]}"; do
                set -- $p
                result="$result${1##*:}=$(db_getval "$2" "$3"), "
        done
        print -- "${result%', '}) at $file:$line"
}
```

Figure 2.12: `ksh` implementation of `gdb`'s `frame` command

- Invoking target functions from the debugger; the nub doesn't support this feature, because a separate evaluation facility can support it better [RH92].
- Examining core dumps; this feature could be supported by writing a nub specifically for browsing core dumps.
- Interrupting a running target.
- Handling signals.

2.8 Portability

The `deet` front end will run on any machine that `tksh` works on. Currently, this includes virtually all UNIX and Windows variants. Graphical debuggers that work consistently under such a variety of operating systems are scarce, and having a constant interface can be a major advantage. Anyone writing code for both platforms can debug without having to learn two environments. People who use two environments tend to stick with a small subset of each environment. Also, it is advantageous that debugging scripts work on all platforms.

Most of the issues of portability are dealt with by the nub implementation. Any machine that lcc has been targeted for can generate a nub at compile time. Also, if a machine can run gdb, `deet` can run using the nub implementation that runs on top of gdb. This means that `deet` can work with any compiler, machine or language that gdb supports. Another option is to rewrite the nub, a task which is usually not difficult due to its minimality.

2.9 Discussion

`deet` demonstrates that it is possible to build a usable debugger with a graphical user interface from simpler components, and, as the `dotty` example illustrates, that the result is more than just the sum of the parts. `deet` also confirms `cdb`'s premise that most of a debugger is machine-independent, and that the fundamental machine-specific debugging facilities can be encapsulated in a small, machine-independent nub interface.

`deet` doesn't have all of the features offered by PC debuggers and UNIX debuggers like `gdb`. But it does provide the most important ones—at a fraction of the implementation cost. `deet` is about 1,500 lines of `tksh` code, and the machine-independent nub and related compiler support (in `lcc`) total around 800 lines of C. These 2,300 lines of code are orders of magnitude smaller than `gdb`'s 150,000 lines and `ddd`'s 90,000 lines.

Programmers interact with most debuggers in the target's source language plus a few debugger-specific commands. For example, programmers throw C expressions at `gdb` to browse the state of a buggy target. The advantage of this approach is that programmers don't have to learn another language to use the debugger. But, as Duel [GH93] demonstrates, exploring a program's state is fundamentally different than writing the program in the first place, and this exploration can be done much more effectively in a higher-level language. `deet` also supports this view; Tcl and `tksh` seem to be better languages for writing debugging code than languages like C and C++. Similar comments may apply to other high-level scripting languages, like Perl.

Chapter 3

Chava

3.1 Introduction

The World Wide Web first started with web servers presenting only static HTML content. Later, Common Gateway Interface (CGI) scripts were introduced to run on web servers to dynamically compose content before presenting them to the clients. Recently, Java applets have been used increasingly on web sites to provide rich user interfaces and perform client-side processing to generate dynamic content. While many web site analysis tools [DBCK98, CK97] are available to analyze the structure of static HTML content, most of them completely ignore the applet code, which by its nature requires software analysis techniques.

Traditional software repositories [SB95, Tho89, CGK97, Dev92, BMG+94] apply reverse engineering [CI90] techniques on the source code to build a central information source for maintaining code in a software system. Repositories are useful to developers as they make it possible to efficiently examine the structure and interaction between components of a system without having to delve through potentially hundreds of thousands of lines of source code. Advanced tools have also been built to perform reachability analysis [CGK97], clustering analysis [MMR+98], selective regression testing [CRV94] and even extraction of light-weight object models [SvG98, JW99].

This chapter presents Chava, a reverse engineering and tracking system for Java [AG96]. The system presented has several noteworthy features:

- **Data Model for both Byte Code and Source Code:** Like Womble [JW99] and some recent Java tools, Chava can work on binary class files directly. However, unlike other tools with a single-task focus, Chava aims to have a *complete* data model (as defined in Acacia [CGK97]) at the selected abstraction level – class member declaration – to support a wide range of analysis and tracking tasks. It gets additional information

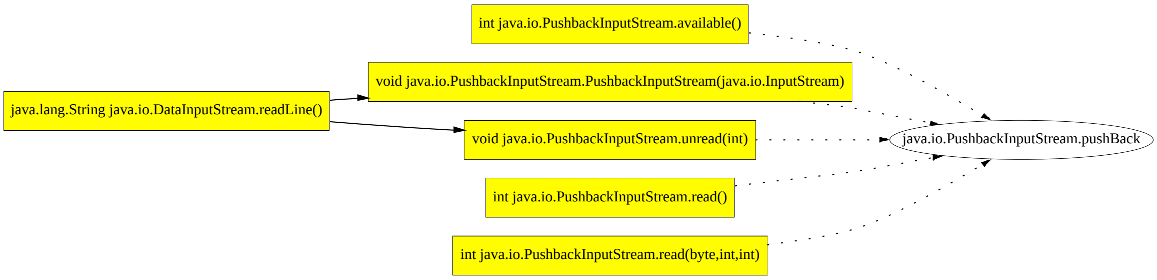


Figure 3.1: A Sample Reverse Reachability Diagram from the Difference Database of JDK1.0 and JDK1.1

(such as line numbers) from source code when it is available.

Analysis using only class files is possible primarily due to properties of the Java language. Java does not have a preprocessor, which means that we do not have to deal with constructs such as macros, include files, and templates, whose information would not be available in an object file. Also, Java is an architecture-neutral language, so its byte code is the same on all machines. This makes it possible to scan through object code in a machine-independent manner to discover relationships in a program.

- **Program Difference Database:** Chava supports differencing of Java program databases. Similar to the work on change detection in Java from University of Waterloo [RKK98], and in the earlier work of *ciadiff* [Che95] for C and Cdiff [Gra92] for C++, Chava allows tools to examine what changes have been made in two different versions of a system. However, the approach is quite different: Chava can take two previously-built databases and create a difference database with minimal efforts.

- **Integration with HTML Analysis:** Chava can analyze web pages along with the embedded Java applets by combining its database with HTML analysis results created by WebCiao [CK97], which also uses an entity-relationship model.

To give a quick idea of the capabilities of Chava, Figure 3.1 shows a sample diagram generated by our tool from the difference database created for JDK1.0 and JDK1.1. The query was

> Show all the methods that referred to any deleted, protected field member in any Java class.

The diagram shows immediately that only one protected field, `PushbackInputStream.pushBack`, was deleted (shown as a white oval) in JDK1.1, and five methods were affected by this change, all in class `PushbackInputStream`. It also showed that all these references have now been removed (represented by dotted edges) in JDK1.1. By doing a reverse reachability analysis for three layers, we see that the method `DataInputStream.readLine`, which refers to two of those methods that used to access the deleted field, is affected by this change as well and should be retested. Note that solid edges indicate relationships that remain in the new version (JDK1.1). Finding correlations between a new software feature and changed program entities and relationships is frequently useful in helping locate problems should they arise after the introduction of the new feature.

3.2 A Data Model for Java

Our Java Data model is based on Chen's entity-relationship model [Che76]. Each Java program is viewed as a set of entities, which may refer to each other. Entities exist for each language construct, such as classes, methods, and fields. Relationships between entities encompass notions such as inheritance and method invocation. This section describes in more detail the composition of the entities and relationships.

A property that our model must satisfy is that of *completeness* as described in Acacia [CGK97]. In order for our model to be complete, it must be the case that if the compilation of an entity A depends on entity B, a relationship between A and B is in the model. We satisfy this condition with one notable exception. In Java, classes can be loaded and methods can be invoked dynamically at runtime using the reflection API [Ref98]. Programs that do this may not satisfy completeness. Completeness allows us to perform analyses such as dead code detection and reachability.

In selecting an appropriate model, a level of granularity must be chosen. Too little granularity will prevent a user from being able to make nontrivial queries. However, too much granularity leads to a database that is too large to handle queries efficiently. Our model handles class member declarations. We create entities for all constructs up to this level of granularity in a program, but do not include information at the level of statements and expressions. That means detailed control-flow analysis or pattern matching on program constructs [PP94] is not available with this level of abstraction.

We will illustrate the model with an example of a simple Java program. Figure 3.2 contains the source code for a set of classes that implements circles and rectangles. The base class `Shape` is extended to implement `Circle` and `Rectangle`.

3.2.1 Entity types

Our model handles the following Java entity types:

```
// file name is Shape.java
package graph;

abstract class Shape implements Cloneable
{
        public abstract double area();
        public abstract double circumference();
        public void printArea() {
            System.out.println("Area:"+area());
        }
}

class Circle extends Shape
{
        protected double r;
        protected static double PI = 3.14159265;
        public Circle(double r) { this.r = r; }
        public double area()
                { return PI * r * r; }
        public double circumference()
                { return 2 * PI * r; }
}

class Rectangle extends Shape
{
        protected double w, h;
        public Rectangle(double w, double h)
                { this.w = w;  this.h = h; }
        public double area() { return w * h; }
        public double circumference()
                { return 2 * (w + h); }
}
```

Figure 3.2: A Simple Java Program

- *class*: Contains declarations and definitions of a collection of methods and fields.
- *interface*: Interfaces are similar to classes, but do not contain definitions. Classes implement the declarations of zero or more interfaces.
- *package*: A set of classes.
- *file*: Source code that contains one or more classes
- *method*: A function that is part of a class
- *field*: A variable or constant that is part of a class
- *string*: Strings that are referenced by methods or fields.

For example, in Figure 3.2, we have the following entities:
Classes: Shape, Circle, Rectangle
Interfaces: Cloneable
Packages: graph
Files: Shape.java
Methods: Shape.printArea, Circle.Circle (constructor), Circle.area, Circle.circumference, Rectangle.Rectangle, Rectangle.area, Rectangle.circumference
Fields: Circle.r, Circle.PI, Rectangle.w, Rectangle.h
Strings: `"Area:"`

3.2.2 Attributes

All entities contain the following attributes: *id*, *name*, *kind*, *file*, *begin line*, *end line*, and *chksum*. The *chksum* attribute is a 64 bit integer that can be used to compare two versions of an entity. This attribute is used when comparing two databases that represent different versions of the program.

The model also includes other attributes that only apply to certain kinds of entities. Table 3.1 summarizes these attributes. They are as follows:

- *parent*: The parent of a method, a field, or a string is its defining class. The parent of a class is the package that contains the class. Packages and files do not have parents.
- *scope*: The scope attribute contains one of private, protected or public. It is used for class entities, which can be public or private, and class members (fields and methods) which can be public, private or protected.

Type	Parent	Scope	Flags	Params	Dtype
Class	package	public,private	abstract,final	N/A	package name
Interface	package	public,private	abstract,final	N/A	package name
Package	N/A	N/A	N/A	N/A	N/A
File	N/A	N/A	N/A	N/A	N/A
Method	class	public,private,protected	abstract,final,native,static, synchronized	params	return type
Field	class	public,private,protected	final,static,transient,volatile	N/A	type
String	class	N/A	N/A	N/A	N/A

Table 3.1: Other Attributes

- *flags*: A set of zero or more modifiers for a given object. For methods, modifiers include abstract, final, native, static, and synchronized. For fields, modifiers include final, static, transient, and volatile. For classes and interfaces, modifiers include abstract and final. Flags are represented in the database as a bit vector.

- *params*: For method types, this attribute contains a list of parameters for the method. This attribute is not used by other entity types. It is primarily used to distinguish among overloaded methods under the same name.

- *dtype*: For method types, this attribute contains the return type of the method. For field types, the attribute contains the type name of the field. For classes, this attribute contains the name of the package it is contained in. The full name of a class is constructed by appending the name and dtype attributes of the entity.

  The dtype attribute is used in these two differing contexts to preserve space in the database. However, dtype is a package name in one context and a type name in the other, so there is no overlap in the possible set of values for these contexts.

For example, let us consider two entities from our example program. The field `Circle.PI` has as its parent the class `Circle`, its scope is `protected`, its flag is `static`, params is empty and dtype is `double`. The constructor for class `Rectangle` has as its parent the class `Rectangle`, its scope is `public`, flags is empty, params is `(double w, double h)`, and dtype is `void`.

In cases where we are working without source files, some entities will be missing values for the begin and end line numbers. Java class files do not include line numbers for definitions of classes, fields and strings. However, bytecode for methods includes line numbers annotations, which we use to find the begin and end line numbers for this entity type. Entities with missing line numbers do not affect the ability to do most analysis tasks on the code.

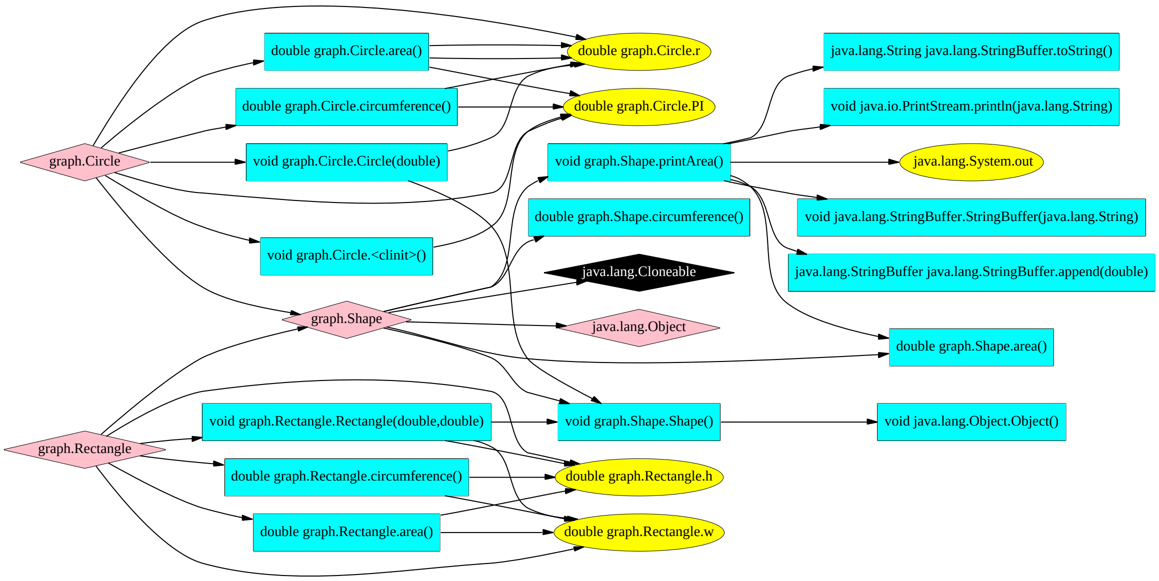


Figure 3.3: Relationships in Example Program

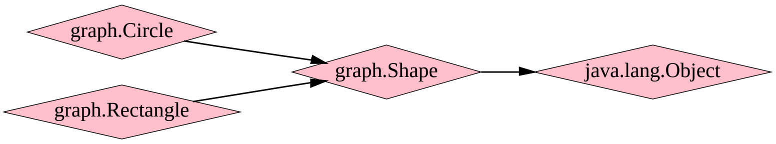


Figure 3.4: Class inheritance diagram

3.2.3 Relationship Types

Our Java data model contains the following relationships:

- *subclass*: Figure 3.4 shows the subclass relationships that exist in our example program. The class `Shape` has `Object` as a subclass, and both `Circle` and `Rectangle` have `Shape` as a subclass. Arrows go from class to superclass in order to show the direction of dependency and maintain the completeness property.

- *containment*: If a field or method A is a member of class B, then a containment relationship exists between B and A.

- *implements*: The implements relationship exists between a class entity A and an interface entity B if class A implements B. In the example, `Shape` implements the interface `Cloneable`.

- *field read*: A field read relationship exists between a method entity A and a field entity B if method A reads field B.

- *field write*: A field write relationship exists when method A field writes to B.
- *reference*: Reference relationships exist between two method entities A and B when A invokes B.

3.3 Java Program Analysis

Once we generate a database for a particular Java application, we can then use a number of supporting tools to extract information about the program. This section starts with examples of basic database queries and program visualization, followed by reachability analysis and program differencing.

3.3.1 Visualization Queries with CIAO

Using a retargetable reverse engineering system called CIAO [CFKW95], a user interface can be generated to graph relationships in a Java program. CIAO takes a specification for a language and generates a set of supporting tools for querying and visualizing databases for that language. Instantiations of CIAO exist for a variety of languages including C [CNR90], C++ [CGK97], HTML [CK97], and ksh.

Going back to our example in the previous section, we can use the Java instance of CIAO to show its class inheritance graph as seen in Figure 3.4. Another query can be made to the database returning all relationships that exist in the application, as shown in Figure 3.3. In the graphs, each entity type is drawn with a different shape, color, etc. as defined in the CIAO specification.

If we want to see all of the members of the class `Circle`, we can perform a similar query, returning all relationships that are of the type *containment*. Figure 3.5 shows the resulting graph.

3.3.2 Reachability Analysis

There are two kinds of reachability analysis, *forward reachability* and *reverse reachability*. With forward reachability, we start with an entity and compute all entities that are in the transitive closure of relationships from that entity. This kind of analysis is useful if we have a large set of classes and methods available to us but only need to use a subset for a given program.

Reverse reachability detects entities which depend on a given entity, either directly or indirectly. This type of analysis is useful if we want to make a change to an en-

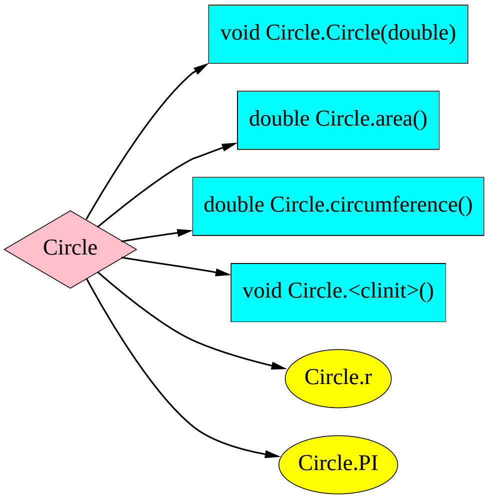


Figure 3.5: Members of class `Circle`

tity such as a method or field and we need to see what other parts of the program will be affected by the change. Figure 3.6 shows reverse reachability analysis on the method `java.lang.Number.intValue`. The graph shows that this method is called in the class `Number` as well as from a method called `subParse` in the class `SimpleDateFormat`. Thus, we have learned that any change to `intValue` will require the methods `subParse`, `byteValue`, and `shortValue` to be either modified or retested appropriately.

Reachability analysis along with a difference database and dynamic traces were also used in TestTube [CRV94] to determine what test cases need to be rerun during *selective regression testing*.

3.3.3 Differencing

Our supporting tools allow two different databases to be compared against each other. Using a set of attributes, all entities in one version of the database are compared against another version. From this, we get a list of added and deleted entities. For entities that occur in both versions, we use the `chksum` field to compare the entities. If the checksum matches, the entities are considered the same, otherwise they are considered changed.

To demonstrate one of the uses of differencing, we have constructed databases for the Java 1.0 standard library and the Java 1.1 standard library. Using differencing, we can easily see what changes were made between these two versions of the libraries. For example, Figure 3.1 showed differencing combined with reverse reachability.

Figure 3.7 shows differencing combined with forward reachability. The original program is our example from Figure 3.2. We compare this to a new version that has removed the method

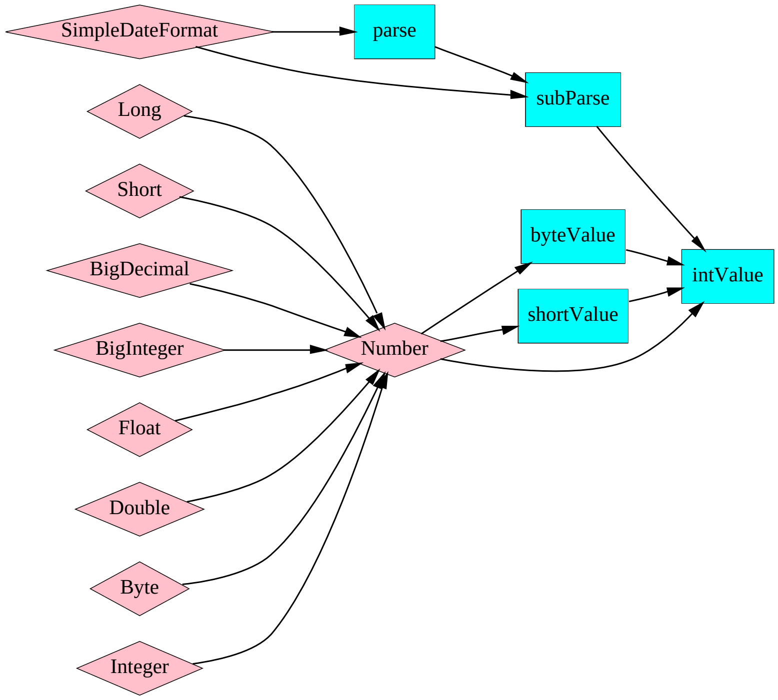


Figure 3.6: Reverse Reachability Example

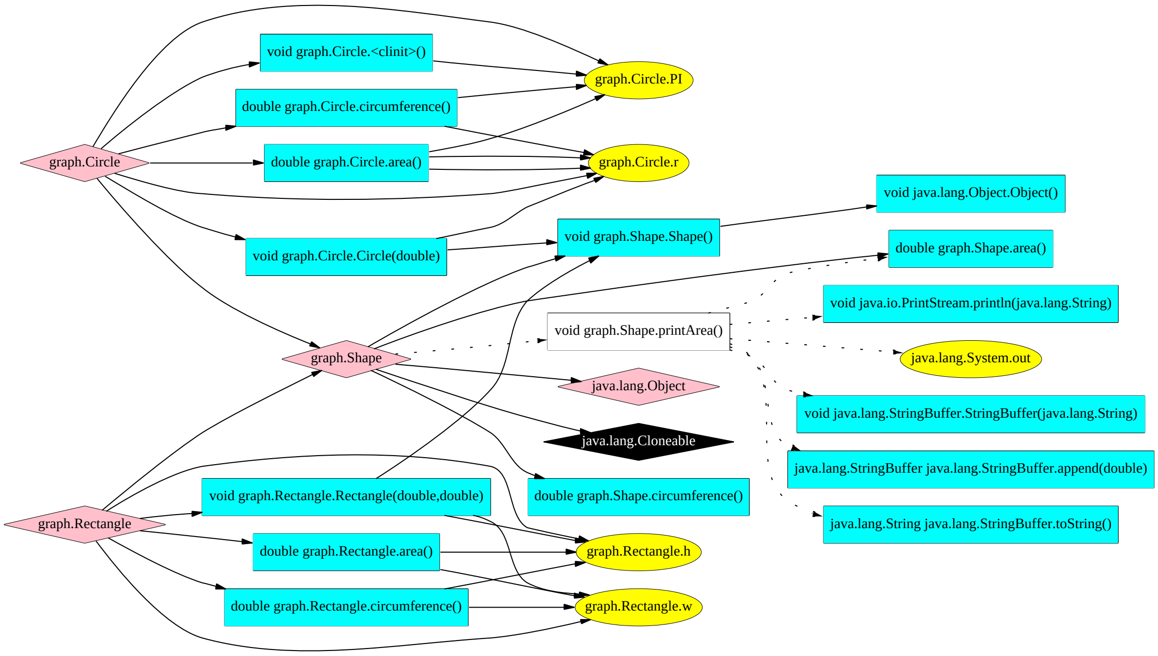


Figure 3.7: Difference Example

`printArea` from the class `Shape`. We view all relationships in the new program showing difference information. The dotted arrows show relationships that have been deleted. The white box shows the member function that has been deleted.

3.4 Applications

This section presents some examples of how Chava can be applied to either software systems or analysis of websites.

3.4.1 Clustering

We have been working with several other researchers to study the use of clustering techniques [MOTU93, HB95, Sch91] on software repositories to discover high level software structures from existing code. We applied Bunch [MMR+98], the clustering tool developed in this purpose, to the Java software repository we created for a proxy server called iPROXY [RCCC99], and obtained the cluster diagram shown in Figure 3.8. The proxy server consists of 3,600 lines of Java code and we were able to produce the cluster diagram, including the process of creating the database and performing cluster analysis, in less than a few seconds. The resulting clusters are consistent with the developer's view on the high-level structure of the code: the right cluster corresponds to a Java CGI component, the middle cluster represents a web server component, and the left cluster represents the agent component of the proxy server. Such a road map is potentially useful for future software maintainers who did not develop the original source code to quickly identify the logical components in the proxy server.

3.4.2 Security

Chava analysis can be useful in detecting potential security flaws in an applet or application. If we download a set of binary Java classes off the web, we can use the results of queries to see which potentially dangerous methods and classes are being used in the program. For example, if we run a query asking Chava to find all methods that invoke a method whose parent package is `java.net`, then we can see if the application is making use of the network API. Also, by analyzing changes in the internal structure of a Java applet, we can find out if the dynamic content of a website has been changed in a malicious and not obvious way.

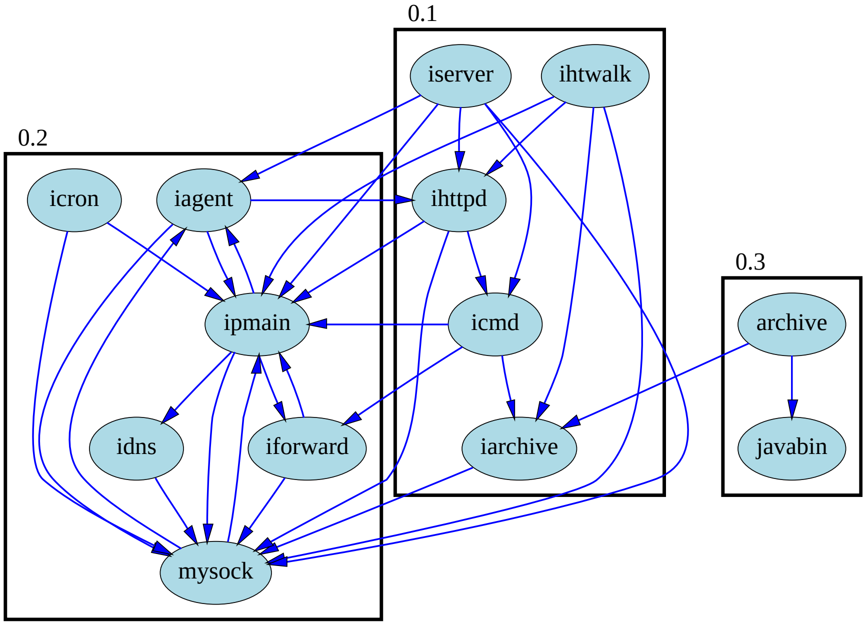


Figure 3.8: A Cluster Diagram for a Proxy Server Written in Java

3.4.3 Object Model Extraction

Similar to Womble [JW99], we can use the extracted information stored in a Chava database to produce object models of software systems. Object models present a graphical abstraction of the interactions between components in an object-oriented program. By examining the definitions of methods and fields in a class, as well as references to objects in methods of a class, we can find one-to-one, one-to-many, and many-to-one relationships between classes. Graphical tools such as CIAO can be extended to draw the model. The architectural view allows observers to more easily see relationships between program components, which is useful for the design, implementation and debugging of software systems.

3.4.4 Software Metrics

Using information such as the number of methods in each class and the number of method calls or field references in each method, we can devise software metrics that suggest how complex a system is. For example, Chava can be used to compute the following Object Oriented Design (OOD) metrics proposed by Chidamber and Kemerer [CK94]: WMC (weighted methods per class), DIT (depth of inheritance tree), NOC (number of children), and CBO (coupling between object classes).

3.4.5 Website Analysis

Our tool supports generation of databases for remote applets. When the URL for a web page is specified to the database generation program, it will find any applet tags and process the class files for each applet in the HTML file.

The ability to reverse engineer a remote applet has important uses. A user may wish to reverse engineer an applet on a remote site to get an overview of what it does or analyze potential security problems (see Section 3.4.2).

Since the author of an applet typically writes the application in one environment and uploads the applet to a server where it is run, there is a possibility that the applet will behave differently in each environment. For example, a different set of classes could be loaded on the server due to its configuration. Our tool eliminates the process of moving the applet and its supporting components to a different machine before doing the analysis.

The CIAO environment works with a variety of languages, one of which is HTML. WebCiao [CK97] is a system for visualizing and tracking the structures of web sites using an HTML repository. WebCiao generates a database from a set of web pages, and uses a suite of tools that are generated with a CIAO specification.

We have implemented a tool which combines repositories of HTML and Java applets. The combined repository gives web developers an environment for viewing all aspects of web content together, both static and dynamic, including document structure and applet interaction.

Figure 3.9 shows an example query for the web page `http://www.att.com`. The query shows the reachable set of entities from this url. The web page contains an applet called `AIG_Flipper`, so a relationship exists between the web page and the applet. Thus, this query draws the set of URLs referred to by the web page and the methods referred to by the applet together.

3.5 Implementation

Our implementation consists of two parts. First, there is a program called `chava` that generates a repository for an application. Second, there is a specification file that provides information about the semantics of the database used by external tools.

Chava is implemented as a C program and is around 3,000 lines of source code. A Java-based implementation was considered, but was discontinued after initial performance results were poor. The C-based implementation runs an order of magnitude faster than our Java version.

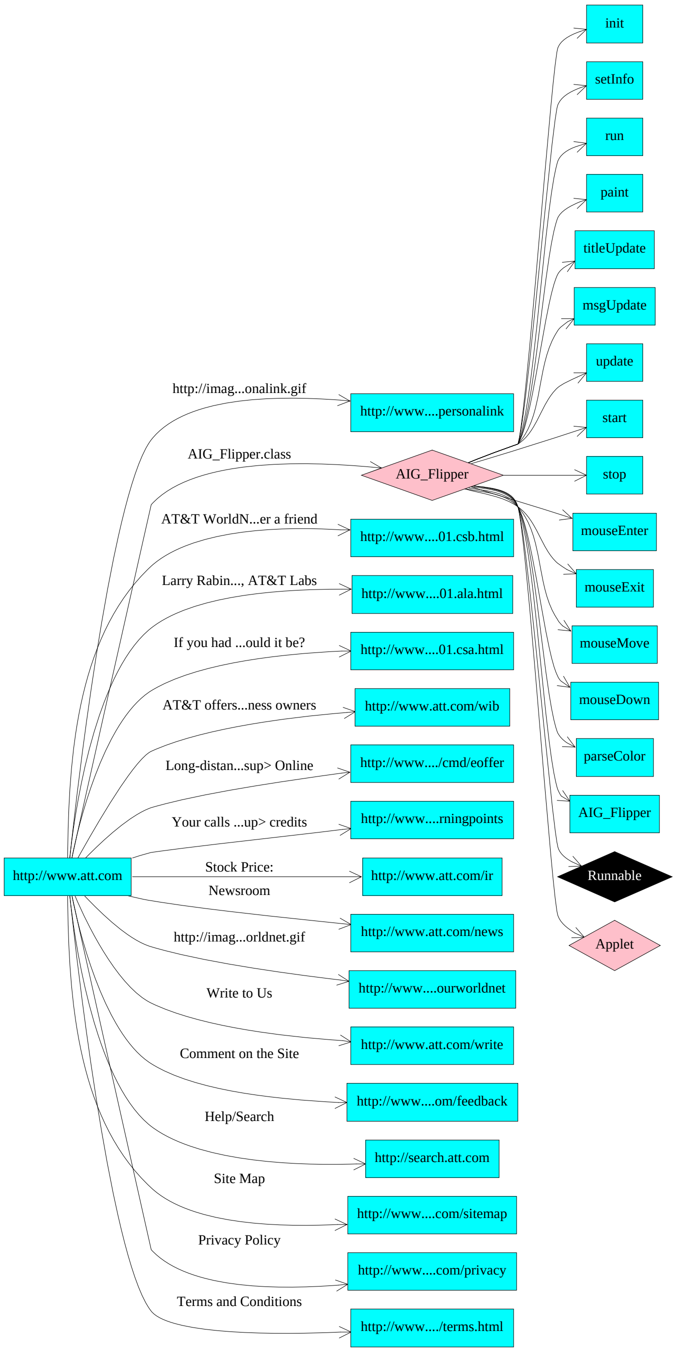


Figure 3.9: HTML and Java Integration

Chava works analogously to a compiler. It first generates incremental databases on either a per-file or per-class basis (depending whether we are working with source or class files), similar to how a compiler generates object files. Then all of the per-class databases are linked together to generate the complete database, similar to the linking phase of a compiler. The linking phase reads in the per-class databases and resolves entities that reference external entities with their definitions.

3.5.1 `.class` Files

Chava is able to generate a database for a class given only access to its `class` file (generated by the compiler). The class file contains enough information about the entities in a class that we can fill in most of the attributes for our model. Class files contain a table of methods, fields, classes and strings. Chava goes through each of these tables and converts the table entries into database entities. To determine the relationships of field access and method invocation, Chava goes through the byte code for each method and picks out instructions that call methods or access fields.

3.5.2 Source Files

Class files do not contain every piece of information that our model provides, but they contain enough to generate a useful database for our tools. Class files are missing some information about line numbers that Chava is able to get by parsing source files. If a source file exists for a processed class, Chava parses it to fill in the missing fields from the database.

If we only have a source file for a class without the compiled class file, Chava invokes the Java compiler to produce the class file and then works with both the source and class files. Thus, Chava is capable of working with only source, only class files, or both.

3.5.3 Archives

Java allows a set of classes to be combined into a single file as an archive. To generate a database for an archive, the archive is first expanded into its source and/or class files. Then, Chava is run on each file and the set of databases is linked to form a single database from the set of classes. The resulting database can subsequently be linked with other applications.

For example, we have produced a database for the Java default class library. The library is quite large and consists of around 2 megabytes of class files. The generated database contains 15,248 entities and 35,012 relationships. This database can be linked in with Java applications if the user wants to do analysis involving the Java class library. For example, reachability analysis on a Java program would reveal how much of the Java library it actually uses.

3.5.4 Java Instantiation of CIAO

The query and visualization subsystem for Java is built by constructing a new instance of the CIAO system. CIAO takes a specification file for a language and generates the supporting tools for querying and visualization automatically. The specification consists of the following parts:

- *Schema.* It enumerates the attributes of entities and relationships. Types for attribute fields include integers, strings, and pointers to other entities.
- *Database View.* This section defines how different entity and relationship entries are viewed in text format. For example, a class entity contains the package name as one attribute and the class name as another. When displaying a class entity, we can specify that the name be displayed by combining these fields.
- *HTML View.* This section defines how to format each of the attributes in the database as HTML, making it possible to display query results in a browser window.
- *Source View.* This section defines the fields that are needed to find the source file and line numbers for a given entity in the database.
- *Graph View.* This section defines how to graphically display entities and relationships. Entities are displayed as nodes, and the specification can define what colors, shapes, fonts, etc. should be used for different kinds of entities. Relationships are represented as edges, and can also be represented with various styles and colors.
- *GUI Front End.* The final section defines the appearance of the graphical user interface.

The specification file is less than 300 lines long. The complete suite of query, visualization, and generic reachability analysis tools are generated from this specification file.

3.6 Performance and Experience

This section looks at the performance of Chava as a function of the number of entities and relationships emitted. It is our goal that Chava scale well with large applications so that it can be useful for real-world software projects.

Program	Source size	Class size	Number of classes
java.*	300,000 lines	1,648,508 bytes	696
swing	211,640 lines	1,340,364 bytes	503
WebDelta	23,951 lines	369,469 bytes	92

Table 3.2: Project information

Program	javac	javap	Compile Database	Link Database
java.*	N/A	1m5.4s	48.6s	22.07s
swing	5m16s	38.29s	30.72s	15.61s
WebDelta	1m8s	8.8s	4.9s	2.52s

Table 3.3: Performance of Chava

Program	Database size	Number of entities	Number relationships
java.*	1,720,326 bytes	15,248	35,012
swing	1,245,338 bytes	9,939	30,371
WebDelta	482,594 bytes	3,807	14,422

Table 3.4: Space Requirements

3.6.1 Speed and Size

To measure the speed of Chava, we have taken a set of applications ranging in size and generated databases for them. To put the numbers in context, we compare the time that Chava takes to the speed of compilation. We also compare the running time of Chava to javap, which is a Java program in the Java SDK that dumps the contents of a class file. javap can be viewed as a lower bound on performance had we used Java to write Chava.

The three programs we performed experiments on were as follows:

1. *java.**: This is a collection of classes that make up the standard library for Java. All classes are in packages that begin with `java.` The set of classes is stored as an archive in zip format, and source is not available. Thus, our tool does not extract some properties from the archive, such as line numbers.

2. *swing*: Swing is another Java archive distributed from Sun. It contains a set of classes that implement a set of user interface components for use with Java. This is a good example of a large application in which source code is available.

3. *WebDelta*: WebDelta is an AT&T project that implements a Java-based version of the WebCiao interface. This is an example of a medium sized software project. Source is available.

The results are shown in Tables 3.2, 3.3 and 3.4. Running times are from a SPARC station running Solaris 2.5. We see in Table 3.3 that Chava is significantly faster than compilation with javac. In fact, Chava is also faster than javap, the Java program that dumps the contents of a class file, despite Chava's increased functionality. This is most likely the case because Chava is implemented as a C program instead of a Java program. Nonetheless, these numbers indicate that the performance of Chava is an order of magnitude better than compilation.

Table 3.4 shows the number of entities and relations generated by each of our test applications. The table also includes the size of the database file, the content of which is database entries in ASCII format. We see that the generated database size is in the order of the size of the class files, which is quite manageable. Size could be significantly reduced if compression were used on the database. The number of entities and relationships is small enough that queries can be performed efficiently.

3.7 Integration with `deet`

Our Java database generation tool is the first instantiation of CIAO to work with object files. Much of the information that we extract from applications is useful for debugging, as our suite of tools help users to better understand programs. We make use of the CIAO tools in `deet`, described in Chapter 2. Some examples of applications that might be useful are: (1) being able to look at the current location of a program inside its call graph and (2) using queries to find a set of line numbers to use as breakpoints (e.g., set a breakpoint at every line that reads field X).

3.8 Conclusion

A growing number of applications are being written which can benefit from tools that assist with software engineering tasks. Our tool makes it possible to work with large applications and perform complex analysis tasks such as reachability analysis and clustering analysis. It also allows visualization of the components of an application to facilitate the understanding of their interaction.

Chava show that existing reverse-engineering tools can be applied to a repository built entirely from object code. The performance analysis of Chava shows that not only can such repositories can be built without source, they can built efficiently. Object code repositories are particularly useful for debugging, as tools that work with such repositories can analyze and visualize programs.

The differencing capability of Chava is particularly useful for debugging. Faulty systems can be compared to properly working systems to isolate where the two versions differ. Differencing can also help with testing, as Chava can detect which parts of a system have introduced new code.

Working with object code instead of source has the disadvantage that we cannot extract information that the compiler has removed (such as optimizations or source comments), but is advantageous in that it allows analysis to be applied to a wider variety of applications, such as applets and legacy systems in which source is unavailable.

We have several tasks planned for future development of our Java repository:

- *Handle errors and exceptions.* In Java, methods may throw exceptions and errors. Each method has a set of exceptions that it is allowed to throw, and others must be caught. Our current implementation does not put information in the database about exceptions and errors. Future versions may create separate entities for them and create relationships between methods and the exceptions/errors they throw.

- *Handle compiler optimizations.* Chava works with class files emitted from Java compilers. However, some compilers optimize code. When source files are not available, Chava has difficulty working with optimized code because entities from the original code may not exist in the compiled class file. If source exists, Chava can always recompile the code without optimizations, but it should be possible to deal with some optimizations at the bytecode level.

- *3D Visualization.* Feijs and De Jong [FJ98] have applied 3D visualization techniques to software systems with encouraging results. 3D provides more ways to represent program relationships, making use of position and color to describe attributes. We plan to apply some of these techniques to Chava databases.

- *Web-Based Reverse Engineering Service.* Instead of installing CIAO and Chava on each user's machine, we are currently creating a web service for developers who would like to share the understanding of a particular program amongst each other. Users will be able to generate graph views using an applet [BML97], run database queries through JDBC, and view source code as HTML or XML [Oas99]. Such a service will allow researchers to freely analyze and experiment with public source code.

Chapter 4

TraVis

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 [ZL96] 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 [BS84] and Zeus [Bro91b], 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.

This chapter 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 of this work 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 4.1 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

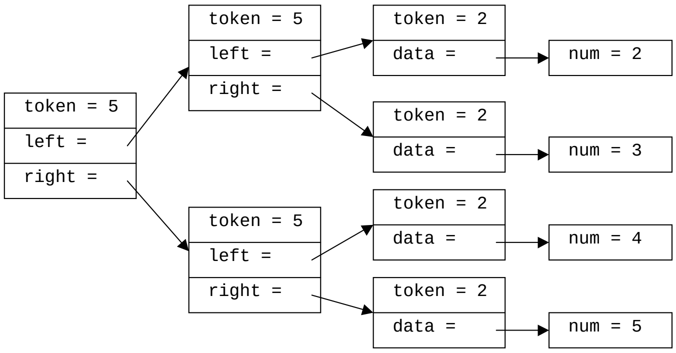


Figure 4.1: Low-level Data Structure Layout

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 4.2, 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 [Moh88] 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 [MS94] 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 attach 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 [RC89], 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 [RCWP92] uses the technique of declarative visualization. However, Pavane is not practical for use as a

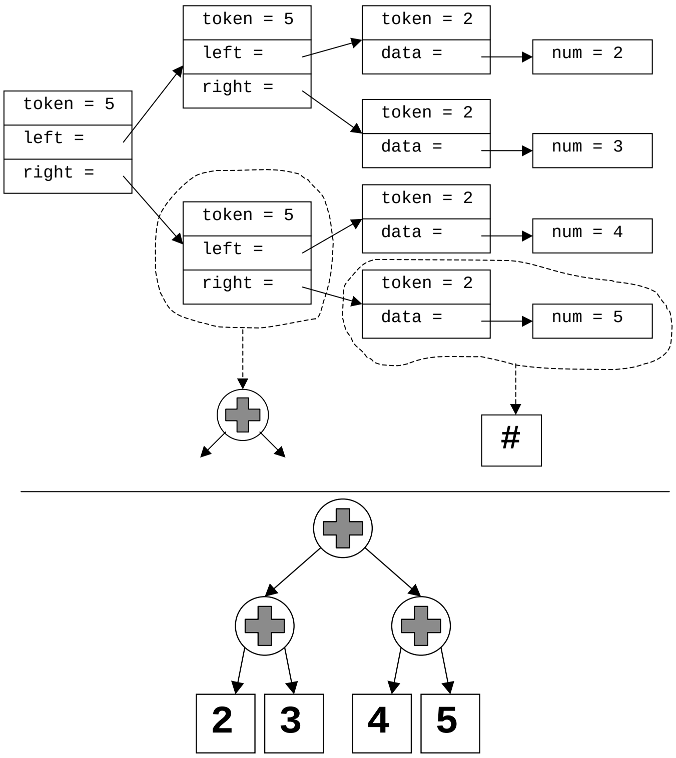


Figure 4.2: Transforming traditional layout into a more abstract representation

debugging system. 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 chapter presents a debugger called TraVis 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 chapter will focus on Java, it is important to note that the techniques are applicable to other languages as well.

TraVis 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 chapter presents an overview of TraVis. 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.

4.1 Foundation

The goal of this work is to produce a debugging environment where data structures can be abstractly displayed. Chapter 2 covered the fundamental operations that the debugger provides such as stepping and setting breakpoints. This chapter focuses 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 TraVis, which accomplishes these goals. The specification language consists of a set of *rules* that define how to draw objects matching a given

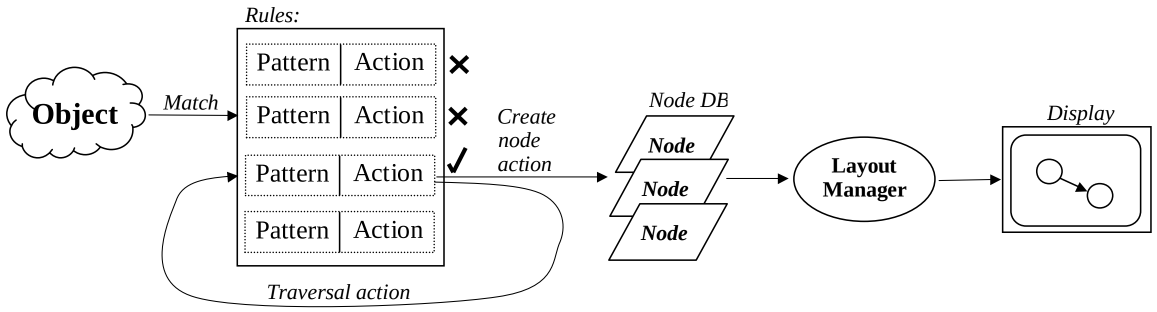


Figure 4.3: System Components

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 4.3 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 (such as scripting languages) or support mutation (such as ML) which are beyond the scope of this thesis.

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:

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

```
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 4.4: EBNF Grammar for Patterns

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 (e.g. `<`) 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 4.5 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.

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

Figure 4.5: EBNF Grammar for Actions

Traversal Database

The nodes generated from the traversal of a data structure are stored into a database before they are rendered. For each node, the database stores its type its set of attributes. Using a database allows us to leverage existing database tools to visualize and postprocess the traversed entities. Nodes in the database are displayed by a separate component, called a *layout manager*, that only needs to know how to communicate with the database and does not have to interact with the target program. Using a data allows us to have multiple displays of the same data structure (for example, a tree can be displayed as a directed graph or as a hierarchical list). It also gives us the ability to use the our layout components for other applications.

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, 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.

4.2 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:

```
public abstract class Expr { }

public class Num extends Expr {
    private int value;

    // Methods not shown
}
```

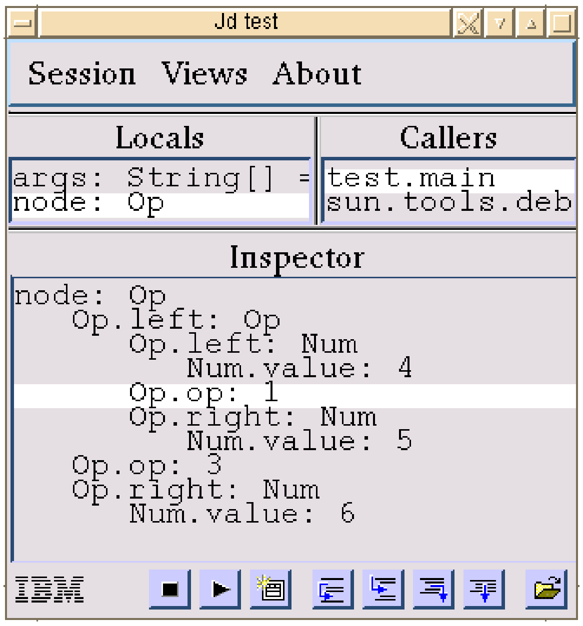


Figure 4.6: Traditional Layout

```
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 4.6 shows how a typical graphical debugger might draw a tree instance without user input [Lie97]. 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 4.6 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"),
    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 4.7. 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.

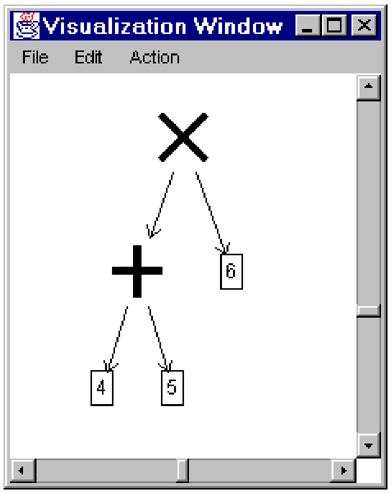


Figure 4.7: Layout with Patterns

```
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 4.8, 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;
```

In Figure 4.9, 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 is shown below:

```
Plane p :
   -> p.point(mode="point"),
   -> p.convexHull(prev=p.convexHull.elementAt(p.convexHull.size()-1),
                   mode="convexHull");

PPoint p when (mode == "point") :
   PointNode(x=p.x, y=p.y,color="green");

PPoint p when (mode == "convexHull") :
   LineSeg(x1=prev.x,y1=prev.y,x2=p.x,y2=p.y,color="blue"),
   (prev = p);
```

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 or from Chava data repositories can be 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

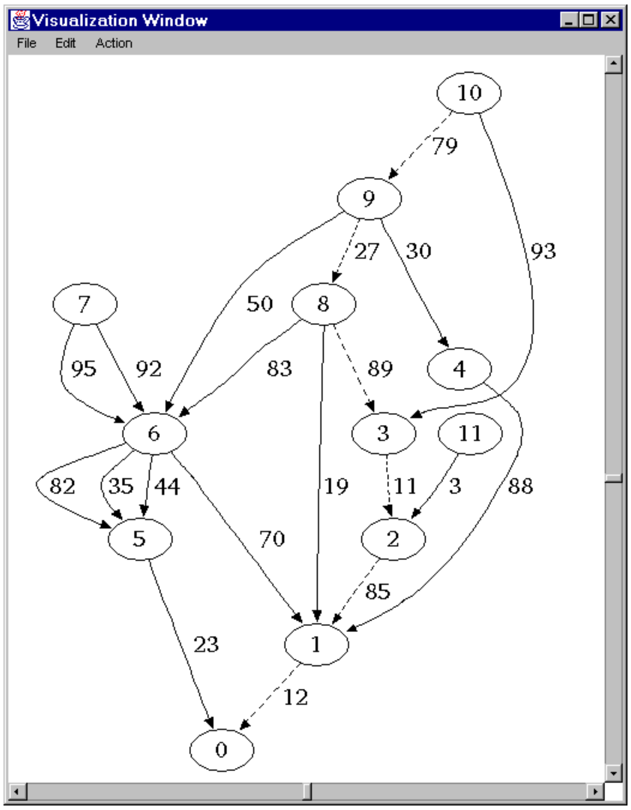


Figure 4.8: Visualization of Longest Path

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. Chapter 5 presents an example of automatic specification generation.

4.3 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

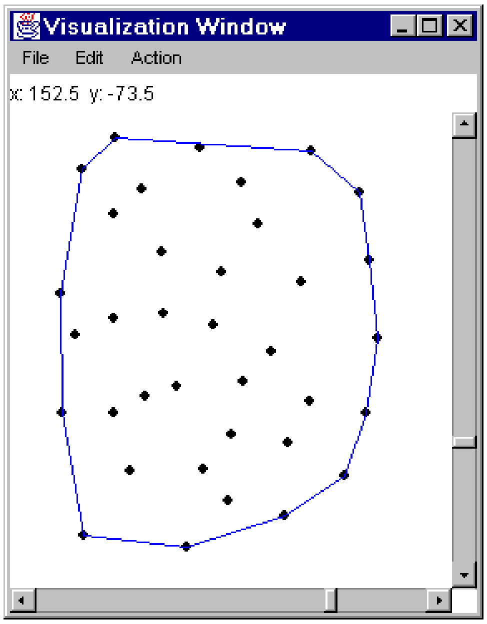


Figure 4.9: Visualization of Convex Hull

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 4.10 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 4.2. The selected nodes become an exact pattern match for that substructure. Or, if multiple groups of nodes are selected, TraVis will use the most specific pattern that matches each selected group. The pattern can then be further refined (by changing a subpattern to be "unrestricted", adding fields, etc.) using the pattern editor interface.

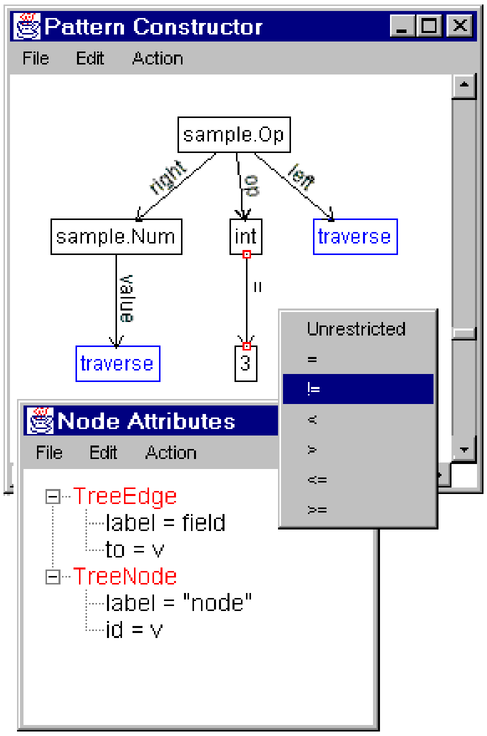


Figure 4.10: Visual Pattern Constructor UI

4.4 System Implementation

This section gives an overview of how the system is implemented. The current implementation is approximately 10,000 lines of Java code.

Debugger Interaction

Our system uses `deet`'s abstract interface to access symbol table information, as described in Chapter 2 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.

`deet` incorporates TraVis by providing a new command called `traverse`. This command accepts an expression to be traversed, similar to the `print` command. Invoking `traverse` brings up a dialog box, giving the user the option of providing the location of a specification file or creating the rules interactively. Upon confirmation, TraVis traverses the specified expression and renders the data structure.

Updates and Redrawing

It is undesirable to redraw an entire data structure each time part of it is changed. Redrawing requires re-traversing every element of the structure, which can be inefficient for large structures 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 in such cases it can revert to redrawing the entire structure. 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, 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

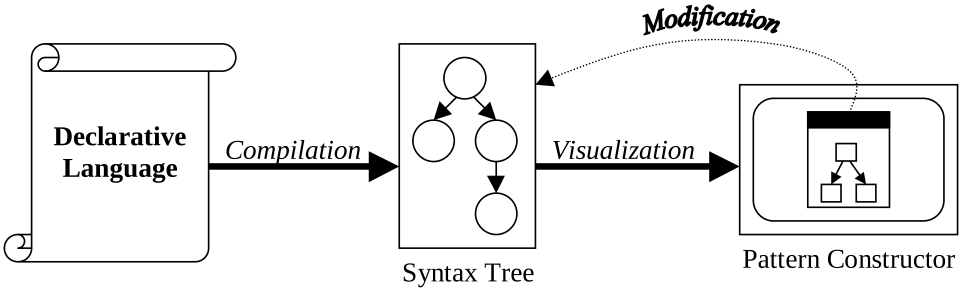


Figure 4.11: Visual Pattern Constructor Design

changes into the object by changing its attributes in the database. 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 to determine what needs to be changed and how. 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 4.11 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.

4.5 Discussion

Preliminary results from the system have been promising. We have used it to put together visualizations for a project called *Zephyr* [WAKS97]. 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. Chapter 5 describes this work in more detail.

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. We have done some user testing to evaluate the ideas found in this system, the results of which are described in Chapter 6.

4.6 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. The visualizations are useful because they present data in a manner that is more natural to the user. Users do not have to reinterpret the display as something that they are more comfortable with. We have implemented TraVis, a debugger that allows transformations to be specified with a pattern-based language. The debugger also provides a user interface to this language.

Another use of a system like TraVis is that it makes it possible to cope with the vast amount of data that real applications contain. Software visualization needs better filters and controls over what information is presented to a user; the many thousands of values in a real program makes it necessary to provide a mechanism to control which data is of interested and which data can be ignored. Psychological studies indicate that people are capable of holding only a small amount of concepts in memory at once [Mil56]. TraVis provides a way to do such filtering through the specification language.

Chapter 5

Visualization of Compiler Trees

In this chapter, we look at a specialized application of abstract data structure visualization. We look into the domain of compilers, which typically work with tree-like structures that represent the program in the intermediate stages of compilation. Our goal was to write tools that make it easier to work with such trees through the use of visualization.

5.1 Introduction

High-quality compilers for a range of modern languages are essential for conducting experimental research in computer architecture, programming languages, and programming environments. Building compilers is often a bottleneck; compiler construction is labor-intensive. Often, nearly complete compilers must be constructed even if the essential components are relatively small parts of the whole. Writing toy compilers for toy languages is insufficient, as the research community demands measurements using established benchmarks written in real programming languages.

Zephyr[1], the National Compiler Infrastructure Project seeks to make it easier to build high-quality compilers. Its goal is to make it possible to build complete compilation systems from pieces, replacing or modifying only those pieces that are relevant to the client researchers. For example, researchers studying global optimization algorithms for C++ would replace or add only their optimizers and would use existing C++ front ends and code generators. Zephyr aims to significantly reduce the costs of doing computer systems research. It should also encourage more researchers to attack computer systems problems and thus

[1]Zephyr is a compiler infrastructure project conducted jointly at the University of Virginia and Princeton University (see `http://www.cs.virginia.edu/zephyr`). This work is supported in part by the Department of Defense under contract MDA904-97-C-0247 and DARPA order number E381, and the National Science Foundation grant ASC-9612756.

increase the rate at which new research results appear.

Reusable components make it easy to build compilers to test new compilation techniques. The components of a compiler communicate with each other through an intermediate representation (IR), which is a description of a program suitable for optimization and analysis. If compiler components can exchange compatible IRs they can inter-operate.

To inter-operate, components need an implementation of an IR and a way to transmit IR values to other components. Since different compiler research groups program in different implementation languages, IRs need to be implemented in more than one programming language. Otherwise compiler components written in different languages cannot inter-operate. A simple way to transmit IR values across different components is to read and write the IR to a file in a standard format. These files are called *pickles*, and conversion to pickles is called *pickling* or *marshaling*.

Unfortunately, many IRs are only described by one implementation in one language. For these IRs it can be hard to separate the abstract structure of the IR from implementation artifacts. Just as a parser implementation is a poor way to describe the concrete syntax of a programming language, a set of data structures is a poor way to describe an IR.

5.2 Abstract Syntax Descriptions

The Zephyr Abstract Syntax Description Language (ASDL) [WAKS97] describes the abstract syntax of compiler intermediate representations and other tree-like data structures. Just as the lexical and syntactic structures of programming languages are described with regular expressions and context free grammars, ASDL provides a concise notation for describing the abstract syntax of programming languages. Tools can convert ASDL descriptions into the appropriate data-structure definitions and functions to convert the data-structures to or from a standard flattened representation. This makes it easier to build compiler components that inter-operate. Existing tools convert ASDL into C, C++, Java, and ML data-structure definitions and conversion functions.

ASDL shares features found in many network interface description languages (IDLs), algebraic data types, and languages such as ASN.1 [ISO87] and SGML [GR90]. Compared to other alternatives ASDL is simple and powerful.

5.2.1 The ASDL Language

ASDL is simple enough to describe with an example. Figure 5.1 is the ASDL description of a trivial programming language.

$$
\begin{array}{rcl}
\textit{stm} & = & \textbf{Compound}(\textit{stm}\ \text{head},\ \textit{stm}\ \text{next}) \\
 & | & \textbf{Assign}(\textit{identifier}\ \text{lval},\ \textit{exp}\ \text{rval}) \\
 & | & \textbf{Print}(\textit{exp}^*\ \text{args}) \\
 & & \textsf{attributes}\ (\textit{pos}\ \text{p}) \\
\textit{exp} & = & \textbf{Id}(\textit{identifier}\ \text{id}) \\
 & | & \textbf{Num}(\textit{int}\ \text{v}) \\
 & | & \textbf{Op}(\textit{exp}\ \text{lexp},\ \textit{binop}\ \text{op},\ \textit{exp}\ \text{rexp}) \\
 & & \textsf{attributes}\ (\textit{pos}\ \text{p}) \\
\textit{pos} & = & (\textit{string}?\ \text{file},\ \textit{int}\ \text{line},\ \textit{int}\ \text{offset}) \\
\textit{binop} & = & \textbf{Plus}\ \ |\ \textbf{Minus}\ \ |\ \textbf{Times}\ \ |\ \textbf{Div}
\end{array}
$$

Figure 5.1: ASDL description

An ASDL description consists of three fundamental constructs: types, constructors, and productions. A type is defined by productions that enumerate the constructors for that type. In Figure 5.1 the first production describes a *stm* type. A value of the *stm* type is created by one of three different constructors **Compound**, **Assign**, and **Print**. Each of these constructors has a sequence of fields that describe the type of values associated with a constructor.

The **Compound** constructor has two fields whose values are of type *stm*. One can interpret the production as defining the structure of *stm* trees which can have three different kinds of nodes: **Compound**, **Assign**, and **Print** where the **Compound** node has two children that are subtrees that have the structure of a *stm* tree. ASDL descriptions allow the specification of a field name to access the values of constructor fields.

Notice that the *binop* type consists of only constructors which have no fields. Types like *binop* are therefore finite enumerations of values. Tools can easily recognize this and represent these types as enumerations in the target language. There are three primitive pre-defined types in ASDL: *int*, *string* and *identifier*; all other types are built from these primitives.

The **Print** constructor of the *stm* type has a *sequence* for the field "args", which is denoted with a "`*`". The qualifier means that the type of a field is an sequence of zero or more elements of that type.

Types can contain *attributes*, which are fields that are present in all constructors. In the *stm* type, an attribute of type *pos* is specified, meaning that the **Compound**, **Assign** and **Print** constructors each contain an additional field "p".

ASDL also provides *product* types, which are productions that define a type that is an aggregate of several values of different types. They are analogous to `structs` in C language.

The *pos* type is an example of a product type; it is an aggregate of the fields "file", "line" and "offset".

Most languages provide the notion of a special distinguished empty value (NULL, nil, NONE). ASDL provides a convention for specifying that certain values may be empty with the "?" (optional) qualifier, as seen in the "file" field of type type *pos*. In this example, our trivial language could set "file" to null if it were parsing from standard input.

It is important to emphasize that ASDL says nothing about how a definition should be translated by a tool into a specific concrete implementation. The description language and external data encoding are fixed; the particular target language interfaces are not. Different tools may produce different language interfaces, as long as the pickle formats used by various tools are compatible. The complete EBNF syntax of ASDL is shown in Figure 5.2.

$$
\begin{array}{rcl}
\textit{definitions} & = & \{\textit{typ\_id}\ \texttt{"="}\ \textit{type}\} \\
\textit{type} & = & \textit{sum\_type}\ \mid \textit{product\_type} \\
\textit{product\_type} & = & \textit{fields} \\
\textit{sum\_type} & = & \textit{constructor}\ \{\texttt{"|"}\ \textit{constructor}\} \\
 & & [\texttt{"attributes"}\ \textit{fields}] \\
\textit{constructor} & = & \textit{con\_id}\ [\textit{fields}] \\
\textit{fields} & = & \texttt{"("}\ \{\textit{field}\ \texttt{","}\}\ \textit{field}\ \texttt{")"} \\
\textit{field} & = & \textit{typ\_id}\ [\texttt{"?"}\ \mid \texttt{"*"}]\ [\textit{id}]
\end{array}
$$

Figure 5.2: Grammar of ASDL. Braces indicate zero or more. Brackets indicate zero or one.

5.3 ASDL Tools

ASDL descriptions are used by a set of tools to assist with various parts of the compiler writing process. We have constructed the following tools that work with ASDL:

- A prototype definitions generator that reads ASDL descriptions and produces data structures definitions in C, C++, Java, and ML
- A code generator that reads ASDL descriptions and produces pickling routines in C, C++, Java,
- A browser-editor that can graphically view and manipulate arbitrary ASDL pickles

5.3.1 Code Generation

From the definitions in Figure 5.1, it is easy to automatically generate data type declarations in target languages such as C, C++, Java, and ML. For languages like C, each type is represented as a tagged union of values. Languages like Java and C++ have a single abstract base class for each type and concrete subclasses of the base class for each variant of the type.

```
abstract public class stm {
    protected int _k;
    public pos p;
    public final int kind(){ return _k; }
    public static final int Compound = 1;
    public static final int Assign = 2;
    public static final int Print = 3;
}

public class Compound extends stm {
    public stm head; public stm next;
    public Compound(stm head, stm next) { ...}
}

public class Assign extends stm { ...}
public class Print extends stm { ...}
...
```

Figure 5.3: Simple translation to Java

Figure 5.3 shows an encoding in Java, which is applicable for many other languages with objects, such as C++ and Modula-3. The *stm* type is represented as an abstract base class. The various ASDL constructors are translated into subclasses of the abstract base class. Each constructor class inherits a tag that identifies which variant of *stm* it is.

5.3.2 Pickles

Since ASDL data structures have a tree-like form, they can be represented linearly with a simple prefix encoding. The `asdlGen` tool generates functions that convert to and from the linear form. A pre-order walk of the data structure is sufficient to convert it to its pickled form. The walk is implemented as recursively defined functions for each type in an ASDL definition. Each function visits a node of that type and recursively walks the rest of the tree.

$$
\begin{array}{rcl}
\textit{asdl\_ty} & = & \textbf{Sum}(\textit{identifier}, \textit{field*}, \\
 & & \quad \textit{constructor}, \textit{constructor*}) \\
 & | & \textbf{Product}(\textit{identifier}, \textit{field}, \textit{field*}) \\
\textit{constructor} & = & \textbf{Con}(\textit{identifier}, \textit{field*}) \\
\textit{field} & = & \textbf{Id}(\textit{identifier}, \textit{identifier?}) \\
 & | & \textbf{Option}(\textit{identifier}, \textit{identifier?}) \\
 & | & \textbf{Sequence}(\textit{identifier}, \textit{identifier?})
\end{array}
$$

Figure 5.4: Abstract Syntax of ASDL in ASDL

An important property of ASDL is that the abstract syntax of ASDL descriptions can themselves be expressed in ASDL. (see Figure 5.4). Since an ASDL description can be represented as an ASDL value, after a parser converts the concrete syntax of an ASDL description into its abstract form, the description can itself be pickled. Other tools can read and manipulate the abstract syntax without any dependence on the concrete syntax. The browser, presented in the next section, is one such tool.

5.4 Graphical Pickle Browser and Editor

The browser is a graphical tool for viewing and editing arbitrary pickled ASDL values. The browser reads in two pickles to do this. One is an arbitrary pickled ASDL value. The other is the pickled ASDL description of all the ASDL types that occur in the first pickle. Given the ASDL description for the first pickle the browser-editor is able to display the first pickle as a hierarchical list or a graphical tree. It allows the user to specify how each kind of node is drawn by allowing the selection of colors, fonts, etc. Trees can be edited using standard cut and paste operations or by creating/modifying nodes. If the user right-clicks on a particular constructor, a pop-up menu will appear allowing the user to change which constructor-type the node should have.

Both the hierarchical list and graphical tree views can be viewed and manipulated at the same time. Changes made in one view of the tree are immediately reflected in the other. This is convenient, as in some contexts it is easier to view with one representation and edit with another.

Upon selecting a type for an edited or newly created node, the browser fills in new nodes for the children of that type automatically. Pickles edited with the browser can be saved as a new pickled value.

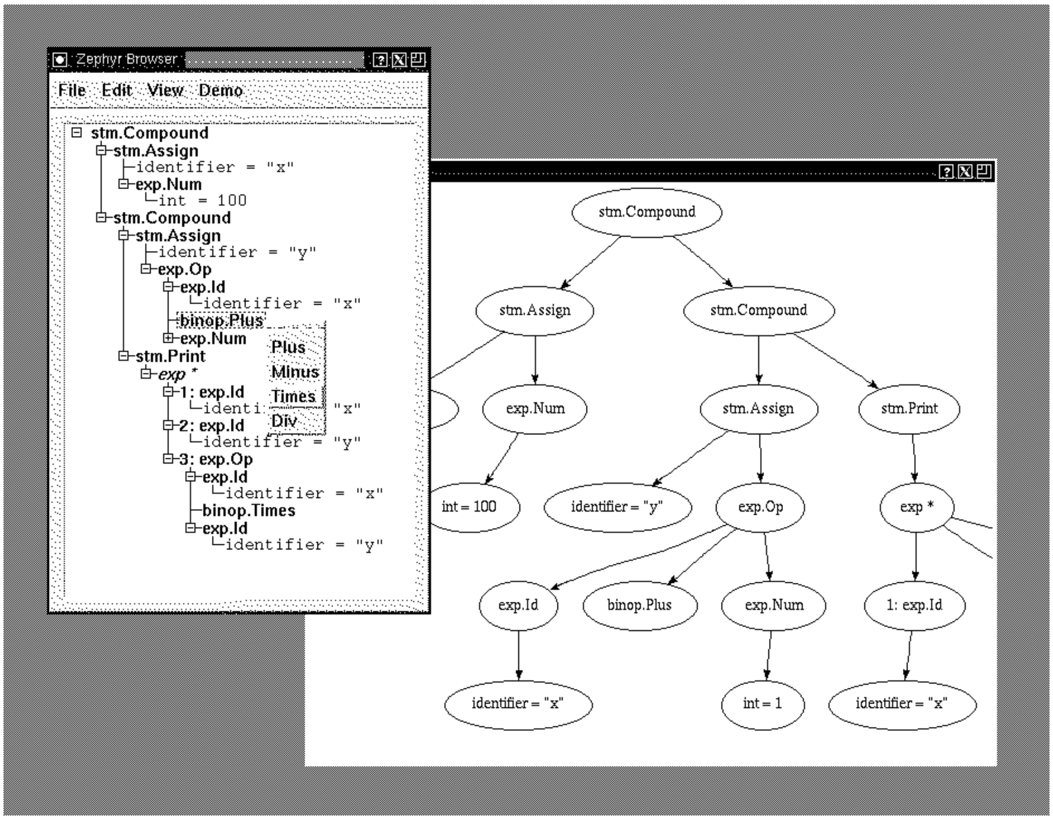


Figure 5.5: Browser-editor for ASDL pickles

Comparing Pickles

One important feature of the browser is the ability to compare two pickles. Multiple pickles can be displayed in the browser at once, and two pickles can be chosen as a pair to be differenced. When the `Diff` menu item is selected, the selected trees are compared using the tree distance algorithm presented in [SZ90]. Nodes that are deleted are shown in red; added nodes are shown in green; modified nodes are shown in blue.

Editing ASDL Types

Since ASDL descriptions are also pickled values the browser can be used to create, edit, or view ASDL type descriptions by manipulating the abstract syntax of ASDL directly. The browser-editor and parser for the concrete syntax of ASDL are two different user interfaces to building the abstract syntax of ASDL descriptions.

$$
\begin{array}{rcl}
\mathit{value} & = & \mathbf{SumVal}(\mathit{identifier},\ \mathit{value}*,\ \mathit{value}*) \\
 & | & \mathbf{ProductVal}(\mathit{value},\ \mathit{value}*) \\
 & | & \mathbf{SequenceVal}(\mathit{value}*) \\
 & | & \mathbf{NoneVal} \\
 & | & \mathbf{SomeVal}(\mathit{value}) \\
 & | & \mathbf{PrimVal}(\mathit{priml}) \\
\mathit{prim} & = & \mathbf{IntVal}(\mathit{int}) \\
 & | & \mathbf{IdentifierVal}(\mathit{identifier}) \\
 & | & \mathbf{StringVal}(\mathit{string})
\end{array}
$$

Figure 5.6: ASDL values specified in ASDL

5.4.1 Large Trees

The browser is able to cope with moderately large pickles. This is an important property, as compilers often work with trees that have many thousands of nodes. Instead of trying to display an entire at once, the browser will pick a suitably sized portion of the tree to show determined by the size of the available window area. Other nodes are not drawn until they need to be.

5.4.2 Stand-alone Browser Implementation

The browser is written in C using the Tcl/Tk library. It manipulates a C version of the ASDL abstract syntax produced by the definition generator from the ASDL description of ASDL. The definitions generator and the parser for ASDL descriptions are implemented in Standard ML, but these two tools easily inter-operate with each other because of the standard pickle format.

When the user edits an object, the browser modifies an abstract representation of generic ASDL values in memory, also generated from the ASDL description in Figure 5.6. The in memory representation is then converted into the correct pickle format. This makes the browser completely independent of the actual details of both the concrete syntax of ASDL or the actual pickling format. As long as the abstract structure of the pickles and concrete syntax stay unchanged, changes to the details of the pickle format or concrete syntax have little impact on the browser.

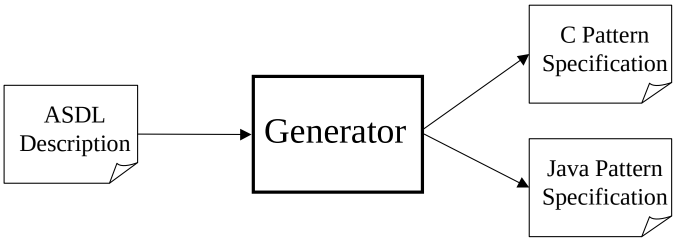


Figure 5.7: Generation of TraVis Specifications

5.5 Implementation for TraVis

The stand-alone browser works well with pickled values, but it is sometimes desirable to work directly with the in-core data structures of a compiler. If we wish to debug a compiler, we don't want to have to pickle our structures each time we need to visualize them. For this reason, we have adapted the Zephyr browser to work as a layout manager form TraVis.

This work also serves as a good test to evaluate the flexibility of TraVis. It shows two properties of the system: First, that TraVis is able to produce the same visualizations that an established tool such as the Zephyr browser already provides. Second, it demonstrates that traversal specifications can be automatically generated as an alternative to the declarative language and the graphical user interface.

As seen in Figure 5.7, our tool accepts an ASDL specification and generates a TraVis specification file using the description. We can generate specification files for any target language, however, currently we only generate Java specifications. The specification files specify mappings between the Java data-type definitions and the tree-nodes to be displayed in the browser. When the specification is applied in the debugger, a database will be generated that the browser uses. The user never sees the specification file, as it is generated automatically.

In order to use the browser as a layout component for TraVis, it was modified in two ways:

- Instead of using its internal data structures (originally holding the pickle) to generate a tree to display, it uses the database of nodes.
- When modifications are made, the browser updates attributes of the nodes. Callbacks have been written to handle the updates by modifying the underlying data structure appropriately.

Users can also use the graphical pattern editor to fine-tune the automatically generated patterns to create more specialized displays of the data, something that was not possible to do in the browser in the stand-alone version.

5.6 Evaluation

The stand-alone version of the browser has been used in the undergraduate compiler course at Princeton University, where students construct a compiler for a toy language from scratch. The browser was used to debug syntax trees and intermediate trees in their assignments. Students in the course programmed their compilers in ML, and were provided with a routine that pickled an IR tree of the toy language into a file. The browser could then be used to view the tree.

Students reported satisfactory results with the browser. A feature that was particularly useful was the tree differencing feature. A student with an improperly working compiler could compare his or her IR tree to one generated by a fellow student's working compiler, and quickly narrow down the problem areas. Since IR trees tend to be large, this was an invaluable time saver.

Some examples of comments collected from students were:

- The browser is invaluable. There is no other way to view IR trees other than to write code to traverse and print them out. The browser provides a convenient way to do all that and moreover allows you to build or modify the data structures in question.

- The browser is easy to use. It is easy to traverse the ASDL data structures. It allows you to modify the data structures that you are dealing with so you can easily experiment and test your algorithms.

- It was very useful. Visualizing the IR makes identifying bugs in the front end an easy task. Without some form of visualization, it is often hard to tell exactly where a bug is in the compiler.

- The best feature, in my opinion, is that the browser enables you to browse the IR in a way that you can quickly go to the branch that you are interested in.

5.7 Conclusions

Declarative languages such as regular expressions and context-free grammars, with tools like `lex` and `yacc`, help popularized the notion of describing the concrete syntax of programming languages formally. Description languages like ASDL will popularize the notion of formally describing the abstract syntax of programming languages and the internal representations of compilers. Our initial experience suggests that ASDL is able to encapsulate the fundamental structures of important data structures in a concise and language independent way.

The browser is capable of reading the pickled data structures generated from ASDL tools as well as debugging in-core structures directly. The visualization of IR trees can be worthwhile for compiler writers, and features such as tree differencing and lazy expansion of values have been beneficial to users of the browser thus far.

It seems appropriate to reuse ASDL descriptions for a wide variety of other tools, such as attribute evaluators, parsers, pattern matchers, and pretty printers. All these systems can benefit from the formalisms that ASDL provides.

Chapter 6

TraVis Evaluation

In Chapter 4, we presented TraVis, a system for debugging that produces algorithm visualizations of program data. This system was developed with the belief that these visualizations assist programmers in fixing their programs. However, the fundamental question of whether TraVis actually enhances the debugging process needs to be addressed.

In this chapter, we evaluate TraVis to gain insight on the use of algorithm visualization, and specifically whether the method of traversal-based visualization is useful to programmers. A study was conducted on a set of programmers, who learned and operated a functioning version of TraVis to debug a set of programs.

There were several questions that this study aimed to answer:

- We wanted to see how long it takes for a programmer to be able to understand and effectively use TraVis. It is important that the time required to learn the system and create visualizations be reasonable; if learning the system takes too long, users will decide that it is easier to resort to traditional methods such as print statements to display the data.

- We wanted to see what effect TraVis has on the way programs are debugged. For instance, one of the potential advantages of traversal-based visualization is that users should not have to understand much of the target's code, only the data structures. This study tested to what extent this claim was true.

- The study attempted to understand how users of TraVis reacted to using declarative specifications. The experiments were designed to see how easy specifications were to write using the declarative language, how long they took to write, and whether users preferred to enter the specifications by writing them directly or by using the graphical interface.

- Our experiments measured the quality of the TraVis user interface. In order for such a system to be effective, users should find the user interface pleasant and easy to use. They should also be able to intuitively figure out features of the system.

- We wanted to know what kinds of visualizations users would produce when they were given the freedom to create their own mappings. Part of the experiment was designed to see if users would produce simple visualizations similar to the boxes-and-arrows program visualizations that can be generated automatically, or if they would produce more sophisticated algorithm visualizations.

6.1 Experimental Procedures

Two pilot subjects were used to test the experimental tasks and procedure; changes were made to to use more descriptive text to explain the system and some of the tasks. The procedure for testing each subject was:

- The subject was given a three page tutorial describing the TraVis. During the tutorial, subjects created a specification file to visualize an actual program step by step by introducing the specification language. The tutorial also described how to use the graphical pattern constructor.

- The subject was given a set of tasks to perform with TraVis. Each task involved invoking the debugger on a given program and producing a visualization of its data. The tasks differed in the types of data being visualized and the method used to produce the visualization. In two of the scenarios, the subject was told how the visualization should look. One was kept open-ended requiring the subject to choose an appropriate mapping.

- The subject was asked to record the amount of time each part of the experiment took. The subject was also asked to write down which portions of the code were examined to produce the visualizations.

- The subject filled out a questionnaire after all tasks of the experiment were completed. The questionnaire is based on QUIS [CDN88], the Questionnaire for User Interaction Satisfaction. There were twenty-nine questions asked about various aspects of the system. Subjects also wrote down general comments about the system.

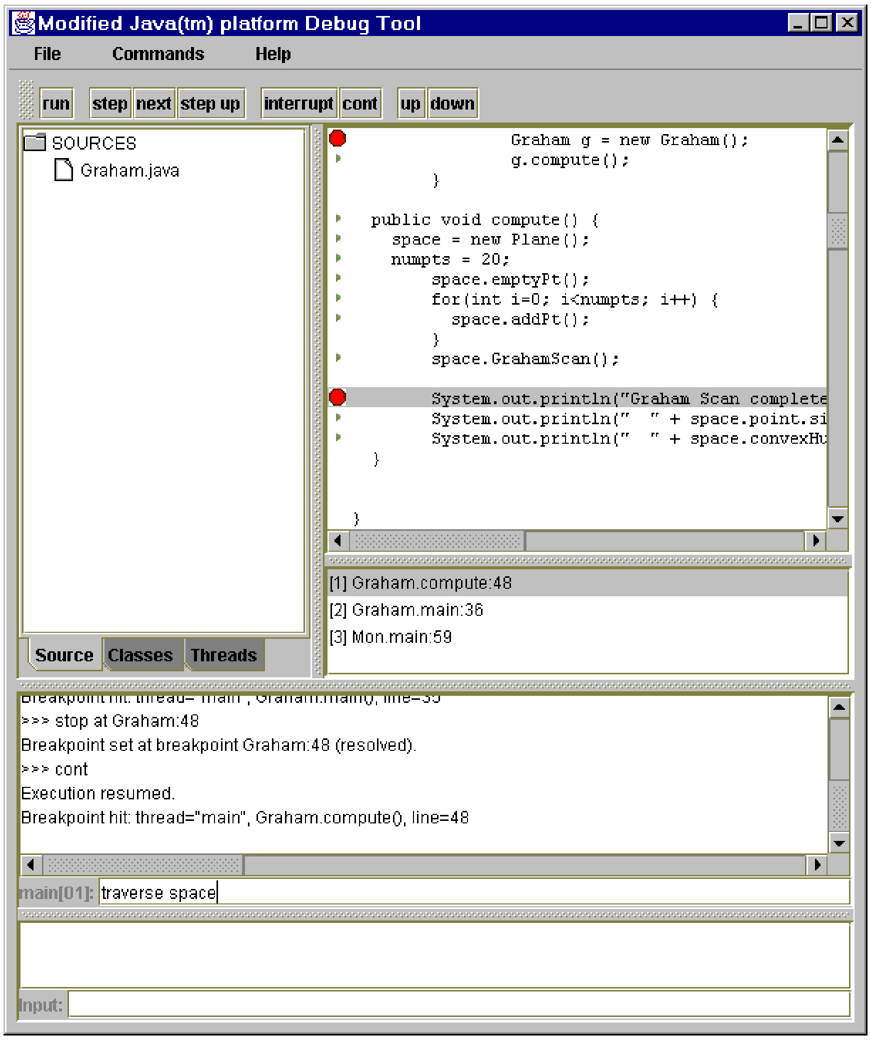


Figure 6.1: Sun's Java 1.2 Graphical Debugger Supporting TraVis

6.1.1 Participants

Eight graduate students in the Computer Science Department at Princeton were the participants. Since TraVis is a debugging tool that requires knowledge of data structures, all participants were proficient programmers. However, some of the participants were not very familiar with Java, and some were not regular users of debuggers. All data was collected anonymously.

6.1.2 Materials

The experiment was conducted on a Windows NT workstation running Java 1.2. TraVis was incorporated into Sun's Java 1.2 graphical debugger, shown in Figure 6.1. The experiment used Sun's debugger instead of `deet` so that it was less likely that artifacts from `deet` would affect the outcome. Sun's debugger resembles typical graphical debuggers such as Microsoft Visual J++ and Borland JBuilder. Sun's debugger was chosen because it was the only debugger available at the time of the experiments that supported the Java 1.2 debugging interface, which TraVis requires.

Subjects were allowed to consult with program source code if necessary. They were asked to note any use of source code. Subjects were also given a short reference guide to TraVis, which contains a list of available layout managers, the types of nodes they support, and the attributes understood by each node type.

Tutorial

Before completing the experiment, subjects needed to get familiar with the system. Subjects were given a short written tutorial, included in Appendix A. The tutorial guided subjects through the creation of a traversal specification for a Java program that computes the convex hull from a set of two-dimensional points. The tutorial first shows how to use the automatically generated boxes and arrows visualization for the selected data structure. Then it shows step-by-step how to modify the visualization to create two-dimensional points which are plotted by a different layout manager. The tutorial covers more advanced concepts of the language and shows how to draw line segments connecting the points on the hull, making use of environment variables and when expressions.

The tutorial also explains the graphical pattern editor. It introduces the user interface and guides the subject through using the constructor to make some refinements to the visualization. Subjects were allowed to ask questions during the tutorial. Although the tutorial was not timed, on average it took between 40 minutes to an hour to complete.

6.2 Tasks

Once the tutorial was finished, subjects were asked to complete a set of tasks using TraVis. For each task, the time required to complete the task was recorded and the specification file was submitted. Each task involved working with a different program, each of which was contained in a different directory. Subjects were given the following instructions:

Sorting

Go to the **sort** directory, which contains a program that sorts an array of random integers. Run `travis Sort`. Write a specification using the 2D Plot Layout component to draw the elements in the array before the sort (by calling `traverse s` in the debugger). Set a breakpoint after the sort and apply the same visualization.

MaxPath

Go to the **maxpath** directory, which contains a program that creates a directed graph with weights for each edge, and computes the path with maximal weight. Run `travis MaxPath`. Write a specification using the Directed Graph Layout component that produces a visualization of the graph. Nodes should be labeled with their number and edges should be labeled with their weight.

Again, run `travis MaxPath`. Produce the same visualization as you did in Part 2 using only the graphical pattern editor.

Compiler Tree

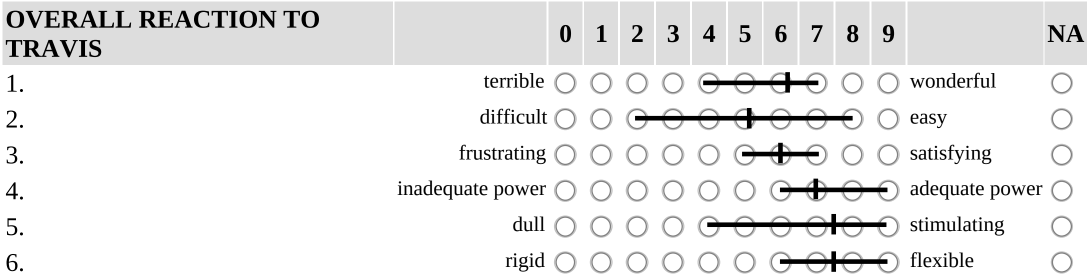
Go to the **tree** directory, which implements simple expression trees. Run `travis Tree`. Show the data structure using the default visualization. Develop a refinement of this visualization that better portrays the semantic meaning of the data. Do this part using the user interface to write the patterns first. Then write the same specification textually with the TraVis declarative language.

6.3 Results

After the experiment, subjects filled out a questionnaire to evaluate various aspects of the system. The questionnaire was modeled after QUIS, a standardized general user evaluation instrument for interactive computer systems. QUIS applies methods of psychological test construction to ensure proper construct and empirical validity of the items.

The questionnaire asked respondents how strongly they felt about a set of statements by marking down their feelings on a 0-9 scale. There were five parts to the questionnaire:

- **Overall Reactions.** This part measured overall reactions to the system. The questions and results are shown in Figure 6.2. The results were mostly positive. On the whole, subjects rated TraVis as stimulating, flexible, and adequately powerful. There were mixed results for level of difficulty; although the average response was towards the middle of the scale (5.2), individual answers ranged from very difficult (2) to very easy (8).



Question	Mean	Std. Deviation	Min	Max
1	6.25	1.30	4	7
2	5.20	2.04	2	8
3	6.00	0.89	5	7
4	7.00	1.10	6	9
5	7.40	1.74	4	9
6	7.40	1.02	6	9

Figure 6.2: Overall Reactions

- **Learning.** This part of the questionnaire asked about how TraVis is to learn. The questions and results are shown in Figure 6.3. This part points to a weakness of the system, as question 7 shows that respondents felt learning to use the system was more difficult than easy (although some respondents gave positive results to this question). Subjects reported satisfaction in terms of remembering the commands and felt that performing tasks was mostly straightforward.

- **Capabilities.** This part showed middle of the scale responses for system speed and reliability, shown in Figure 6.4. This was not surprising, as TraVis is currently more of a research prototype than a production quality system. Some subjects uncovered bugs in the software that were previously unknown. We expect more positive responses in this area as the system becomes more robust.

  Speed was also problematic. However, the system has not yet been tuned for performance and we expect speed to be adequate in future versions. In informal interviews with respondents, the speed problems were mostly in user interface, not in the traversal of data. Some of these problems can be attributed to the immaturity of Sun's user interface and remote debugging libraries.

- **Functionality.** This part asked about key aspects of system functionality, shown in Figure 6.5. The first question in this section was probably the most important of the survey, asking if TraVis makes debugging easier. The result of this question was encouraging, as the average response was 6.75 with little variance. With improved user

LEARNING		0	1	2	3	4	5	6	7	8	9		NA
7. Learning to operate the system	difficult	○	○	○	○	○	○	○	○	○	○	easy	○
8. Exploring new features by trial and error	difficult	○	○	○	○	○	○	○	○	○	○	easy	○
9. Remembering names and use of commands	difficult	○	○	○	○	○	○	○	○	○	○	easy	○
10. Performing tasks is straightforward	never	○	○	○	○	○	○	○	○	○	○	always	○
11. Supplemental reference materials	confusing	○	○	○	○	○	○	○	○	○	○	clear	○

Question	Mean	Std. Deviation	Min	Max
7	4.40	1.85	3	8
8	5.00	2.10	3	8
9	7.20	1.33	5	9
10	6.20	1.72	4	9
11	6.00	2.10	4	9

Figure 6.3: Learning

SYSTEM CAPABILITIES		0	1	2	3	4	5	6	7	8	9		NA
12. System speed	too slow	○	○	○	○	○	○	○	○	○	○	fast enough	○
13. System reliability	unreliable	○	○	○	○	○	○	○	○	○	○	reliable	○

Question	Mean	Std. Deviation	Min	Max
12	5.00	1.67	2	7
13	4.60	2.42	2	8

Figure 6.4: System Capabilities

FUNCTIONALITY		0	1	2	3	4	5	6	7	8	9		NA
14. Travis makes debugging easier	strongly disagree	○	○	○	○	○	○	○	○	○	○	strongly agree	○
15. Using the graphical pattern editor is easier than writing specification files	strongly disagree	○	○	○	○	○	○	○	○	○	○	strongly agree	○
16. If I used Travis, I would visualize data through the graphical pattern editor	strongly disagree	○	○	○	○	○	○	○	○	○	○	strongly agree	○

Question	Mean	Std. Deviation	Min	Max
14	6.75	0.43	6	7
15	4.75	3.49	0	9
16	6.00	2.12	3	8

Figure 6.5: Functionality

interface, results could be even stronger. The second question asked if creating rules with the graphical pattern editor is easier than writing specification files. Interestingly, results were mixed. About half of the respondents felt that the GUI was better and half felt the declarative language was better. This is an important result, as it shows that supporting both methods is useful. The strong response to the third question of this part shows that even those who prefer writing specifications would use the GUI when using the system. This indicates that an important area for future work is on improving the pattern editor.

- **Usability.** The last part of the questionnaire asks a standard set of usability questions, shown in Figure 6.6. Results were mostly a little above the middle of the scale. Of note, there was positive response to question 23 (*I feel comfortable using this system*), negative response to question 24 (*It was easy to learn this system*).

6.3.1 Time

On average, the time required to complete each task of the experiment took approximately the same amount of time, in the range of 16-19 minutes. Deviations were high, as some subjects took less than ten minutes for each part while others took more than twenty. Figure 6.7 summarizes the results.

There was negligible difference in the time it took to create specifications with the user interface compared to writing specifications. Although we conjectured that the user interface would be faster, results did not support this. The most likely explanation for this can be attributed to the user interface of the pattern constructor, which users noted was sometimes “quirky” and “confusing.”

PLEASE RATE THE USABILITY OF TRAVIS.		0	1	2	3	4	5	6	7	8	9		NA
17. Overall, I am satisfied with the ease of completing the tasks in this scenario	strongly disagree											strongly agree	
18. Overall, I am satisfied with the amount of time it took to complete the tasks in this scenario	strongly disagree											strongly agree	
19. Overall, I am satisfied with how easy it is to use this system	strongly disagree											strongly agree	
20. I can effectively complete my work using this system	strongly disagree											strongly agree	
21. Using Travis in my work would enable me to accomplish tasks more quickly	strongly disagree											strongly agree	
22. I am able to efficiently complete my work using this system	strongly disagree											strongly agree	
23. I feel comfortable using this system	strongly disagree											strongly agree	
24. It was easy to learn to use this system	strongly disagree											strongly agree	
25. I believe I became productive quickly using this system	strongly disagree											strongly agree	
26. The interface of this system is pleasant	strongly disagree											strongly agree	
27. I found it easy to get Travis to do what I wanted it to do	strongly disagree											strongly agree	
28. This system has all the functions and capabilities I expect it to have	strongly disagree											strongly agree	
29. Using Travis in my work would increase my producticty	strongly disagree											strongly agree	

Question	Mean	Std. Deviation	Min	Max
17	6.60	1.50	4	8
18	5.60	2.06	3	9
19	5.60	1.50	3	7
20	6.00	1.22	4	7
21	5.75	1.09	4	7
22	6.67	0.47	6	7
23	7.00	1.22	6	9
24	5.00	2.37	2	9
25	6.25	1.64	5	9
26	6.20	0.75	5	7
27	6.20	1.94	3	9
28	6.25	1.09	5	8
29	5.50	1.66	3	7

Figure 6.6: Usability

Part	Avg. Time (*min.*)	Std. Deviation	Shortest	Longest
1	16.00	4.77	7	20
2	18.20	6.11	8	25
3	16.25	6.61	9	27
4	18.60	8.21	10	34

Figure 6.7: Time required to complete each part of the experiment, in minutes

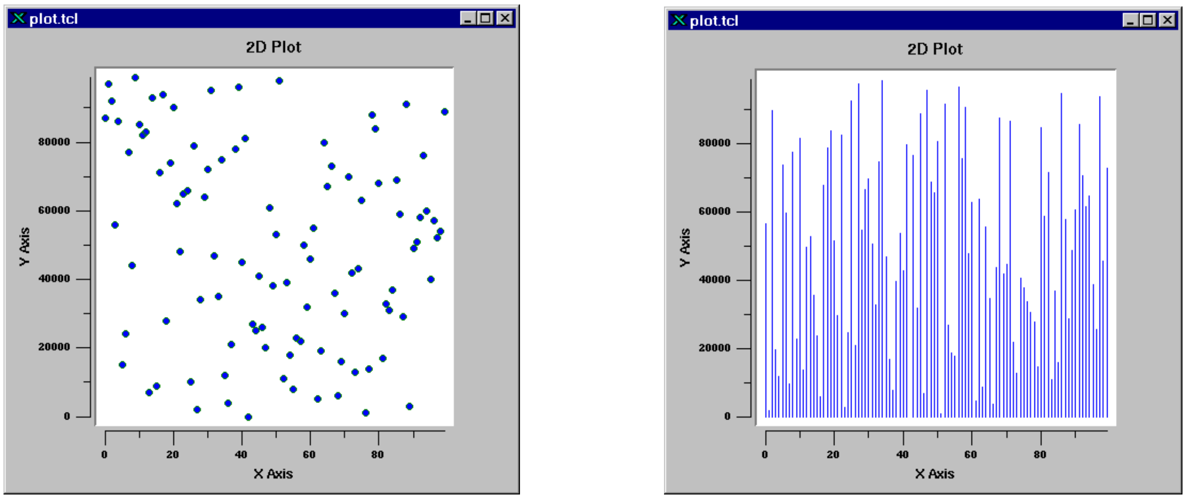


Figure 6.8: Visualizations of an Integer Array

6.3.2 User-created Specifications

In the final task of the experiment, described in Section 6.2, users were allowed to create a specification of their choice to visualize a syntax tree for a toy language. The specifications turned in for this part varied to a small degree. Subjects were asked to refine the default visualization, and all participants simplified the trees by excluding the display of redundant nodes. The other refinement that was expected was to display operation nodes in the tree – which were internally denoted with integer values – with the name of the operation. However, about half of the subjects did not make this change, and left the operations denoted with integers. Figure 6.9 shows some examples of the visualizations that users created for this task.

6.3.3 Comments

Figures 6.10 and 6.11 list some of the comments subjects return on their questionnaires. Based on these remarks as well as oral comments, the overall response to TraVis was encouraging. The consensus was that once familiar with the system it was very effective and

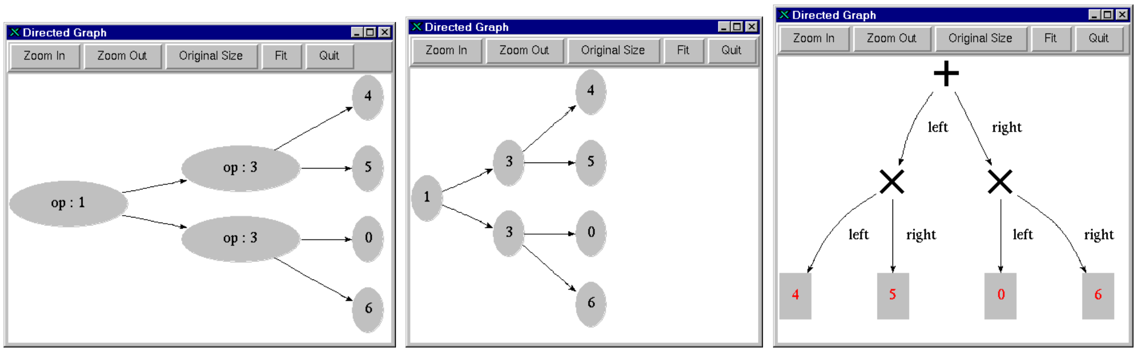


Figure 6.9: Visualizations of the Compiler Tree

useful. The biggest hurdles were that it took a little while to get used to and the user interface needed improvements.

Positive Comments
Pretty pictures fast without to much work
Once comfortable, should be quite easy to write specs
Flexible - different ways to visualize data
Easy to learn/handle
Ability to see my data-structures the way I want

Figure 6.10: Positive Comments from Questionnaire

Negative Comments
Not very robust
Editing specs using the GUI is difficult
More documentation is needed (e.g. when/how to use self, etc)
Some operations take quite a bit of time
Options in pop-up windows sometimes confusing

Figure 6.11: Negative Comments from Questionnaire

6.4 Discussion

The results from the experiments show strength in the declarative language. Due to its simplicity and structure, subjects were able to learn the language and produce useful visualizations quickly. The tutorial and experiments lasted about two hours, which was enough time to cover most of the language and write a set of specifications. It was surprising to see that subjects almost equally preferred the specification language and the graphical pattern editor. We expected the graphical editor would be more desirable, as it is easier to learn. However, the power and flexibility of the specification language was attractive enough to some participants that they preferred using it.

It was interesting to find that user interface played a major role in the evaluation of TraVis. In order to improve results, time needs to be spent making it possible to put together visualizations easier through the interface. One idea that could greatly improve the interface would be the use of *wizards*. Wizards are user interface components that make it possible to rapidly complete a specific tasks by asking a set of questions. When the questions are answered, the wizard sets up all of the lower level detail to accomplish the task which can be edited later. For TraVis, an example wizard could create a 2D plot by asking the user what type they are interested in, and which fields of the type should be used as the x and y coordinates.

The experiments themselves could be enhanced. Instead of having users complete a given set of tasks in a set period of time, it would be useful to collect data from programmers using TraVis in day to day use over a longer period of time. Another improvement that could be made to the experiments is measuring the difference in productivity between TraVis and a traditional debuggers. A control group of subjects could be asked to fix a set of bugs, and the results could be used to determine if it is possible to find more bugs and fix them faster with TraVisḞinally, the questionnaire could be improved by asking more specific questions about the tool. It would be useful to know exactly which parts of the language and user interface are strong and which could be improved.

Chapter 7

Conclusions

This thesis has presented systems that facilitate debugging at a higher level of abstraction. The `deet` debugger lays groudwork for the design of various debugging tools. It also introduces a high-level debugging language, `tksh`, allowing users to write scripts that debug beyond the level of source code statements and the memory heap. Chava lays groundwork for the design of debugging tools as well, only it is focused on static content whereas `deet` is focused on dynamic content. Chava extracts program information into a database, allowing a suite of tools to visualize relationships in the target program. Chava can compare Databases to help track down potential errors in a revised program.

TraVis extracts program data into a database using user-supplied traversal specifications, allowing layout managers to visualize the data structures of a program abstractly. TraVis uses the groundwork from `deet` and Chava, as seen in Figure 7.1. TraVis receives information about types in the target program through the information contained in Chava databases. It reads and writes to the target's heap using `deet`'s nub interface.

There are two principal contributions of this thesis:

- The role of the debugger is to allow users to work with the low-level information contained in a compiled program in a higher-level, more understandable manner. Traditional debuggers typically display only source code listings and heap representations of data. In this thesis, we have increased the level of abstraction for both code and data.

- We have shown ways in which improved design of graphical debuggers leads to simpler, more powerful debugging tools.

Each of these contributions deal with a different notion of abstraction in the debugger. The first kind of abstraction is in the display of information, while the second kind of

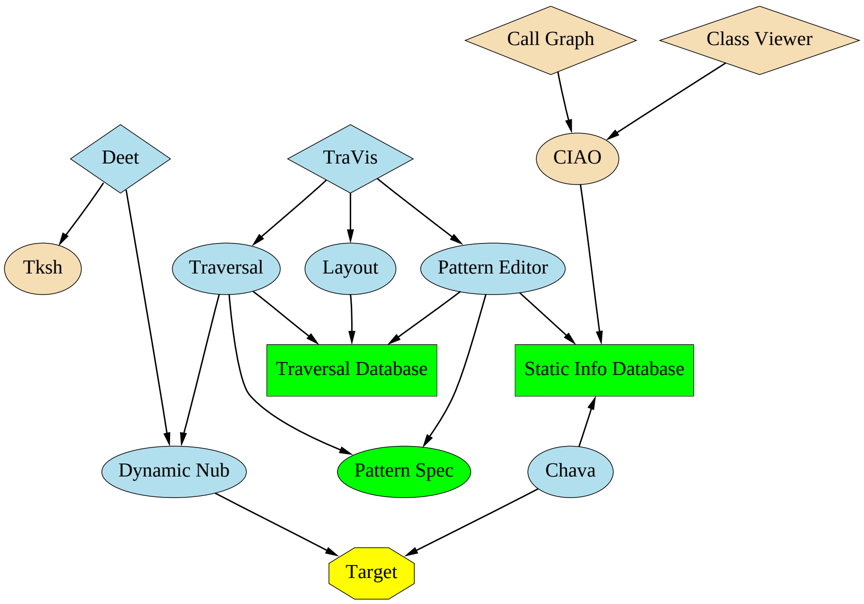


Figure 7.1: Interaction Between Components

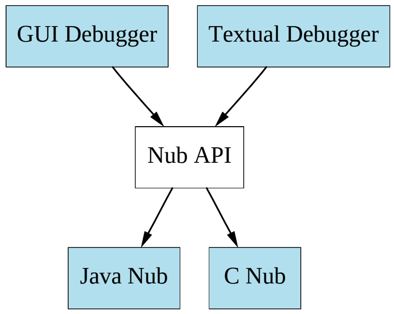


Figure 7.2: Deet abstractions

abstraction is in choosing appropriate design. We discuss each in this chapter.

7.1 Abstraction in Design

One important kind of abstraction is in design. By choosing appropriate ways to break down functionality into modules, along with creating appropriate interface for these modules, we benefit in several ways. All of the systems presented here have paid careful attention to design, which gives them attractive properties.

Let us start with the design of Deet. The separation of machine dependencies into a nub module that works distinct from the interface of the debugger gives us two advantages. First, we can use multiple implementations of the nub with the same debugging front end, making

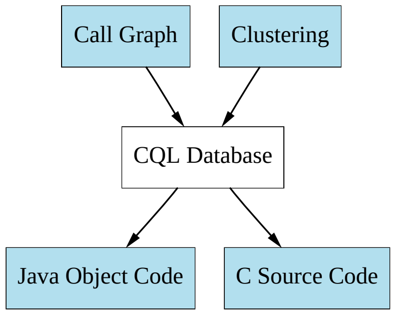


Figure 7.3: Chava abstractions

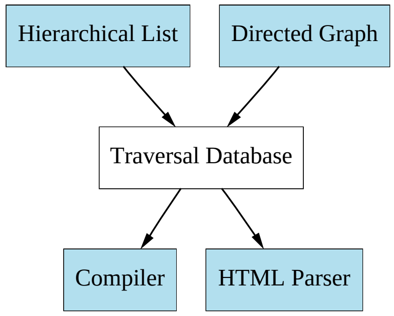


Figure 7.4: TraVis abstractions

it possible to use the same debugger with a variety of languages and systems. By designing the nub interface to be minimal, it does not require much effort to adapt the debugger to work with a new language or environment. Second, we can use the same debugging nub with a variety of debugging front ends. Our Java nub can be used to create a graphical debugger, a textual debugger, or a visualization system such as TraVis.

With Chava, our design uses a simple relational database and schema to describe a Java program. The form of the database is the same for other languages, such as C and C++, which makes it possible to use the same analysis tools on a variety of different databases. The same tool that generates a class hierarchy from C++ source code will work with Java object code. Also, we can write new analysis tools without dealing with the cumbersome details of the source language. If we want to write a new analysis tool for Java, we could use the existing Java database and avoid examining the details of class files and byte code.

Finally, TraVis uses the same design principle of Chava for data visualization. When a traversal specification is applied to a data structure, it produces a database of abstract data entities. A separate module is responsible for turning the database into a visual display. This allows us to have multiple displays of the same data structure (for example, a tree can be displayed as a directed graph or as a hierarchical list). It also allows us to use the same database visualization tools on a variety of applications. The same layout component that

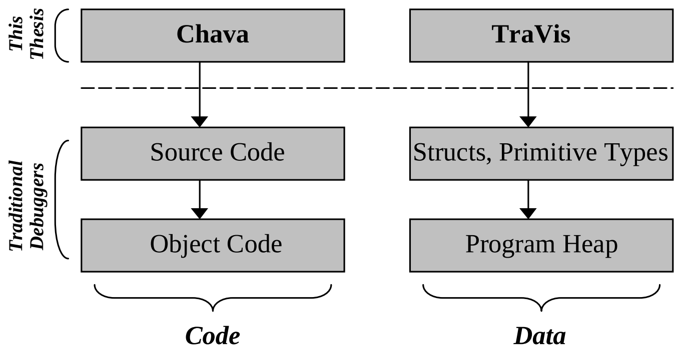


Figure 7.5: Abstraction in Debugger Display for Code and Data

draws trees for a database generated from a compiler will work with a database generated from an HTML parser.

7.2 Abstraction in Display

Debuggers are visibility tools. The task of traditional debuggers is to work with the machine instructions of a program in a more abstract manner by showing corresponding source code. The user is more comfortable thinking about a program in terms of the source code, although machine language is actually run. Debuggers support this line of thinking by mapping source code line numbers to instruction addresses, and by mapping symbol names to memory locations.

We have applied this role of the debugger, which serves to present information more abstractly at a higher level, to other aspects of the debugger (see Figure 7.5). Taking the abstraction of program code a step further, we have devised the Chava tool. By mapping a program's object code into a database of information, tools can display a program at a higher level by looking at abstract relationships between components of the code. The visualizations of class hierarchies, call graphs and reachability relationships are another way of looking at program text. These views are at an even higher level than the source code listing a typical debugger uses.

Analogous to the abstraction of machine code, we have written a system that works with a program's runtime data. With the TraVis tool, users can map the low-level heap representation of data structures to a higher-level algorithm visualization. Traditional debuggers are capable of turning the data in the heap of a program into strings or box-and-arrow diagrams by using the type information emitted by the compiler. TraVis takes this a step further by allowing even higher-level displays of this data.

Abstractions are useful in debugging systems for several reasons. One such reason is that they present data in a manner which is closer to the way the user thinks about it. The user does not have to reinterpret the display into something that they are more familiar with. Another use of abstractions is that they make it possible to cope with the reality of most systems, namely that applications contain a lot of data. A good debugger needs to effectively handle such data. Not only must tools address performance and scalability (as seen in Chava), they must present information in a manner that does not overload the user. Often, looking at hundreds of thousands of lines of code is not as helpful as looking a module dependency graph. Looking at a linked list of thousands of structures is not as helpful as plotting the data in those structures onto a 2D plot.

7.3 Future Work

- **More languages.** 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++.

- **Better UI.** Experiments showed that user interface was a good area for improvement in TraVis. Enhancements to the interface will decrease the learning curve and show the viability of traversal-based visualization in production quality debuggers.

- **More complex data structures.** TraVis has only been tested on simplistic data structures. We will look at complex software systems and see what problems arise when writing traversal specifications for the data structures in these systems.

- **Animation, 3D, Audio.** The current implementation of TraVis only provides static pictures. This work can be extended to provide smooth animations found in algorithm animation systems. In order to accomplish 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 values of elements in an array and swapping two elements. Thus, we need to provide a way to specify such supplementary information to layout managers, similar to what algorithm animation systems do.

- **Integration of database visualization tools.**

  Tools such as Tioga [ACSW96] and DEVise [LRB$^+$97] allow users to build sophisticated visualizations of relational databases by providing a way to interactively compose primitive database operations together. Basic operations such as selections, joins and projections can be successively combined, and these systems are capable of handling large amounts of data. These tools are well suited for use on the traversal database output by TraVisẆe plan to experiment with applying the tools to create software visualizations using our infrastructure.

- **Automatic detection and visualization of software patterns.** Similar to the Womble project [JW99] project, we can use databases extracted from either Chava or TraVis to automatically detect software patterns in systems. For example, we can detect container classes by looking at the signatures for each of the methods.
- **Visualization of execution paths.** We can apply the traversal specification language to visualize program execution patterns. Specifications would be used to identify sequences of function calls in the program which could be monitored for interestig events. The visualizations would simplify the types of displays seen in Jinsight [PLVW98].
- **Web-based repositories and visualization.** Instead of installing CIAO and Chava on each user's machine, we plan to create a web service for developers who would like to share the understanding of a particular program amongst each other. Users will be able to generate graph views using an applet [BML97], run database queries through JDBC, and view source code as HTML or XML. Such a service will allow researchers to freely analyze and experiment with public source code.

7.4 Summary

This thesis promotes using debuggers to look at high-level information in programs. To support this claim, we built and tested functional systems. `deet` lays a simple and extensible infrastructure for writing high-level debugging tools, and allows user scripting. Chava supports program visualization of object code, and TraVis provides algorithm visualization of data structures using a new model of software visualization called *traversal-based visualization.* We have emphasized the importance of abstraction, both in the context of debugger design and in the visualization of software.

Appendix A

TraVis Tutorial

TraVis is an add-on to traditional debuggers that allows you to visualize your data structures. Here, we are using a Java debugger from Sun that has been modified to support a new command called *traverse*. This command has the same syntax as the *print* command.

Data structures are drawn using what is called a *traversal specification*. This is a file that contains information about how data structures are displayed and traversed. Specifications can be supplied as text files, or they can be supplied through a user interface.

A.1 Default Visualizations

To demonstrate how TraVis works, we will show how to visualize an application that computes the convex hull from a set of points. Let's look at the primary class of this application, called **Graham**. The main routine of this class calls a method called compute, which creates an instance of the class **Plane** called *space*. The method `GrahamScan` is invoked on *space*, which stores its results in the object.

In order to produce our visualization, we need to know what the data structure **Plane** looks like. We can see this by having TraVis draw a default visualization with boxes and arrows of this data structure. To do this, set a breakpoint to the line just after `GrahamScan` is called (click on one of the lines with a `println` statement, right-click, select "Set Breakpoint"). Press or type "cont" (not "run" to get to the breakpoint. When the breakpoint is reached, type in the following command to the debugger (at the `main[01]:` prompt):

```
traverse space
```

A dialog box will appear (note that due to Java bugs it may take a few seconds to appear, and it appears in back of the debugger window, so you will have to move out of the way). Accept the default options hit the OK button. A few seconds later, two windows will appear: the

TraVis control panel, and a window showing the drawn data structure. The data structure will be scaled to fit in the window; you can zoom in to see the detail.

We will modify this automatically produced visualization to draw the points onto a 2D canvas. Save the specification to a file by clicking on the "Save Rules" button in the TraVis control panel. Using any text editor, view the specification file. The specification file will have two rules: a rule that specifies how objects of class **Plane** are drawn, and a rule that specifies how objects of class **PPoint** are drawn.

A.2 Specification Files

Let's examine the specification file in detail. The first rule draws a **Plane** object. To the left of the colon is a *pattern* to match. In this case, by writing `''Plane v''` we state that this rule should only be applied to instances of the class **Plane**. The name of this rule is *v*. On the right side of the colon, we have a list of *actions* that are to be applied upon the match. There are two kinds of actions: *node-actions* and *traverse-actions*.

A *node-action* specifies an entity that will be displayed in our visualization. The line:

```
TreeNode(label = "Plane", id = &(self))
```

creates a node that is of the type *TreeNode*, and assigns it two attributes: *label*, which is text that will be drawn inside the node, and *id*, which is a value that can be used to identify the node. A separate part of TraVis called the *layout manager*, will use this information to actually draw the entity. *self* is a reference to the object that was matched. Using the `&` operator returns a unique id for any object, so assigning the `id` attribute to `&(self)` gives this attribute a value specific to the matching object.

The other kind of action is a *traverse-action*, which specifies other objects to be drawn. In the **Plane** rule, we traverse `self.point`, `self.convexHull`, `self.height`, and `self.width`. Specifying these objects will prompt TraVis to look for appropriate rules that match these objects and apply their actions.

TraVis includes some built-in rules that are not shown in the file. There are default rules for drawing primitive types, such as **String**s, **int**s and **float**s. There are also built-in rules for drawing arrays and (in the case of Java) **Vector**s.

A.3 Modifying Specifications

Now we can edit our specification to draw the point in the data structure. First, remove all *node-actions* (every *TreeEdge* and *TreeNode*). This should leave only the *traverse-*

actions, which are the lines that specify fields to be further traversed (such as `self.width`, `self.height`, etc.). Second, remove all *traverse-actions* from both rules except for `self.point`, which is in the rule for **Plane** (this is the list of input points we are interested in). Third, remove the rule for **int**, since no field of type **int** is being traversed anymore. Finally, add a new node-action to **PPoint**:

```
PointNode(x=self.x, y=self.y)
```

This will create an on-screen entity of type *PointNode*, and set its x attribute to the field `PPoint.x`, and it's y attribute to the field `PPoint.y`. After you modify the specification, make sure that each action is separated with a comma and contains a semi-colon at the end.

Save the specification file. Now, we will layout the new visualization. In order to do this, we first need to change the *layout manager* from the Directed Graph boxes-and-arrows style to a two-dimensional canvas. Click on "Layout Properties" in the TraVis control panel and select "2D Plot" for the Layout Type. Hit OK, and click on "Update Layout" You will see a canvas with a set of points.

A.4 Environment Variables

We now turn our attention to drawing the points on the computed hull. The hull is stored as a vector of points in the field `Plane.convexHull`. We will draw these points differently than the other points, with a different color, so they stand out to the user. Thus, we need two rules for **PPoint** that will be applied in different scenarios. In order for TraVis to be able to determine which rule to match, we need to further constrain our patterns based on external information, namely which part of the class we are traversing.

This is achieved with environment variables. Environment variables are variables that are created and used during the traversal of a data structure. They are set with actions that are enclosed in brackets. For example: `[foo = "bar"]`. Environment variables are global; however, a variable can be introduced whose scope is valid only during a specified traversal by setting the environment values in a traverse action. For example: `-> baz(foo="bar")` will set a value for `foo` that is only used during the traversal of *baz*.

In this example, we will use a variable called `mode`, which will be set to a value depending on which field in **Plane** we are traversing. For input points, we apply the rule only when `mode` is set to `"point"` and for points on the hull we apply the rule one when `mode` is set to `"convexHull"` The rules for each are:

```
PPoint p when (mode=="point") :
```

```
        PointNode(x=p.x, y=p.y, color="green");

PPoint p when (mode=="convexHull") :
        PointNode(x=p.x, y=p.y, color="blue");
```

When we traverse `Plane.point` and `Plane.convexHull`, we set `mode` as follows:

```
Plane p : [mode="point" : p.point],
        -> p.convexHull(mode="convexHull");
```

Replace the rules in your specification with the ones listed above and save the specification. Select "Update Layout" from the TraVis control panel. Hull points should now be drawn in blue. Now, we can make a small change to draw line segments connecting point on the hull. We will create on-screen entities of type *LineSeg*, which contain four attributes for the starting and ending x and y coordinates. We will need our traversal to keep track of the previous point in the hull so that each hull point is drawn connected to the previous one. This can be done with a global environment variable, which we will call `prev`. Our modified rule for matching hull points will be:

```
PPoint p when (mode=="convexHull"):
    LineSeg(x1=prev.x, y1=prev.y, x2=p.x, y2=p.y, color="blue"),
    [prev = p];
```

The environment variable `prev` is updated after each hull point is examined. We also have to set an initial value for `prev`. This should be the last point in the `convexHull` vector, since the last point in the hull should connect to the first. Thus, we add the following environment variable assignment to the **Plane** rule before `convexHull` is traversed:

```
[prev = p.convexHull.elementAt(p.convexHull.size()-1)]
```

Again, save the specification file and choose "Update Layout" The hull should be drawn appropriately.

A.5 Graphical Pattern Editor

Patterns can also be edited with a graphical pattern editor. From the control panel, select "Edit Rules" A list of rules will be displayed in a list widget (beware that it takes several moments to come up on the screen). Each entry corresponds to an entry in a specification file. By right-clicking on an entry, a popup menu will appear with a set of options that will affect the selected rule. One of the options is **Edit**, which will bring up a window graphically displaying the pattern and actions for the rule (the pattern will be on the left, and the actions will be on the right). The pattern can be manipulated by selecting nodes and right-clicking to see a menu of options. You can further constrict a pattern by adding fields to the pattern, which can have values set that must be matched. On the right-hand pane of the editor, traverse actions are displayed blue, and node-creation actions are displayed in red. You can view the attributes of a node by expanding it in the action list (click the dot to the left of the name). Actions can be manipulated by right clicking and choosing an appropriate menu item. Once you have made modifications to a rule, you can select “Apply” or “Close” from the “File” menu to update the visualization.

Appendix B

Tksh

B.1 Introduction

Tcl[Ous94] is a general-purpose scripting language which can be reused for a variety of different tools. It is designed to be extensible by allowing operations to be performed through a C library interface, such as adding new commands and manipulating variables. The most well known and one of the most useful extensions to Tcl is Tk, which allows graphical user interfaces to be developed quickly and easily.

One of the reasons that Tcl/Tk is widely used is that developing user interfaces with high-level scripting languages can often be easier and faster than using a low-level language such as C or C++. High-level languages are able to hide many bothersome details such as storage management from the user.

Tcl/Tk is not the only user interface scripting language available for UNIX systems. The Desktop KornShell, which is part of the Common Desktop Environment (CDE), is one alternative. dtksh allows users to develop Motif user interfaces using the new KornShell command and programming language (ksh93)[KNK96]. The goal of Tksh is to combine the strengths that dtksh has for scripting with the strenghts that Tcl/Tk has for developing user interfaces.

Several reasons make KornShell a reasonable choice for an underlying scripting language:

1. *Backward Compatibility* - A large number of users have programmed in either the Bourne Shell or KornShell. It is possible to write scripts without having to learn a new language. Furthermore, existing shell scripts can be easily adapted to provide a user interface.

2. *Conformance to Standards* - KornShell conforms to the IEEE P1003.2 and ISO 9945-2

shell standards. It deals with the issues of internationalization such as error messages, collation order, and international character classes used in pattern matching.

3. *Language Features* - ksh93 has a full set of programming constructs such as functions, floating-point math, associative arrays, loops, and pattern matching. With such a rich set of features, having to spawn other processes is infrequent, making ksh93's performance good.

4. *Interactive Features* - ksh93 is designed for interactive use with features such as command line editing and job control. This is not typically true of other scripting languages. It also has features that make it easy to work with processes and files, yet is not tied to UNIX and runs on other operating systems.

Tksh allows unmodified Tcl scripts to be evaluated in the KornShell interpreter, providing compatibility with the large number of existing Tcl/Tk scripts. Thus, transition from Tcl/Tk to Tksh is easier since scripts do not need to be rewritten. The execution of a Tcl script shares state with the ksh93 interpreter, which means that ksh93 and Tcl scripts can be used together, sharing the same function and name space. For example, a Tcl script can specify a ksh function to be executed as a callback for a Tk event. Similarly, a Tksh script can specify a Tcl procedure to be executeded for a callback. It is therefore possible to use widgets written in Tcl (such as an open file dialog box) in a Tksh script.

Tksh can also be more convenient than Tcl/Tk for some tasks. For interactive scripting, such as debugging with `deet`, features such as command line editing and job control are useful. Scripts that do a lot of work with processes and files are easy to construct with Tksh.

B.2 Using Tksh

When Tksh is run, all of the the commands from Tk exist as shell builtins, and ksh93 can accept Tk commands with ksh syntax. For example, the "*Hello, world*" program can be written as follows:

```
pack $(message .b -text "Hello, world")
```

Scripts can be written that take advantage of features in ksh93. For example, the following command can be used in an interactive shell to create a button that always displays the current directory. When the button is clicked, the current directory will be set to the user's home directory:

```
function selectFile
{
        [[ -d "$1" ]] && cd "$1" > /dev/null
        [[ -f "$1" ]] && ${EDITOR-${VISUAL-emacs}} "$1"
}

function PWD.set        # Discipline called when directory changed
{
        nameref dir=.sh.value           # .sh.value is value being stored
        .f.ls.list delete 0 end
        .f.ls.list insert end .. *
        [[ ${VisitedDir["$dir"]} != "" ]] && return
        .f.dirs.list insert end "$dir"
        VisitedDir["$dir"]=1
}

typeset -A VisitedDir   # Associative array
set -o markdirs

pack $(frame .f)
pack $(frame .f.dirname -relief raised -bd 1) -side top -fill x
pack $(frame .f.ls) $(frame .f.dirs) -side left
label .f.dirname.label -text "Current directory: "
label .f.dirname.pwd -textvariable PWD
pack .f.dirname.label .f.dirname.pwd -side left

scrollbar .f.ls.scroll -command ".f.ls.list yview"
listbox .f.ls.list -yscroll ".f.ls.scroll set" -width 20 -setgrid 1
pack $(label .f.ls.label -text "Directory Contents") -side top
pack .f.ls.list .f.ls.scroll -side left -fill y -expand 1

scrollbar .f.dirs.scroll -command ".f.dirs.list yview"
listbox .f.dirs.list -yscroll ".f.dirs.scroll set" -width 20 -setgrid 1
pack $(label .f.dirs.label -text "Visited Directories") -side top
pack .f.dirs.list .f.dirs.scroll -side left -fill y -expand 1
bind .f.dirs.list "<Double-1>" 'cd $(selection get)'
bind .f.ls.list "<Double-1>" 'selectFile $(selection get)'

cd .
```

Figure B.1: Listing of Sample Program

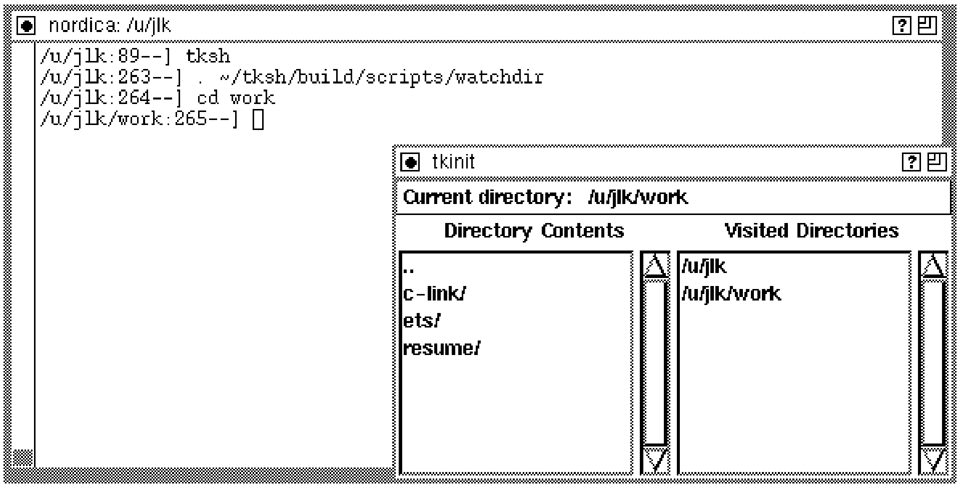


Figure B.2: Screen Dump From Sample Program

```
pack $(button .b -textvar PWD -command cd)
```

When using Tksh, all Tcl commands are available and can be used by prepending `tcl_` to the corresponding Tcl command. For example, the Tcl command `info` has the name `tcl_info` in ksh93. The exact Tcl names are not used because many of the commands have conflicting names with ksh93 and UNIX commands. For example, Tcl contains commands such as `while` and `if`, which conflict with the corresponding keywords in ksh93. Although using Tcl commands in a ksh93 script is infrequently necessary, doing so can be more convenient for a programmer familiar with Tcl. The following example uses the Tcl command `trace` to create a window that always displays the contents of the current directory:

```
pack $(listbox .list -width 20)
tcl_trace var PWD w trace_pwd
function trace_pwd
{
        .list delete 0 end
        .list insert end *
}
```

A discipline could be used instead of a trace in the above example by eliminating the `tcl_trace` command and changing the name of `trace_pwd` to `PWD.set`.

Figure B.1 has the listing of a small Tksh program. The script displays a window containing the contents of the current directory as well as a list of visited directories in an interactive shell. Double clicking on the name of a file will invoke the editor on the selection; double clicking on a directory will set the current directory to the selection. If the a command is typed into the shell that causes the current directory to change, the window is updated. Sample output is shown in Figure B.2.

As the listing of the program indicates, code to set up a user interface with Tksh tends to look similar to Tcl scripts with different quoting rules. The parts that differ the most are in the implementation of the script's functionality.

Evaluation of Tcl Scripts

Tcl code can be embedded into a ksh93 script by using the `source` built-in command. By issuing the command `source` [*filename*], the file *filename* (or standard input if no file is specified) is parsed and interpreted as a Tcl script. The sourced Tcl script has access to and can change the current state of the ksh93 interpreter.

Most existing Tcl/Tk scripts can be evaluated using `source` with no modification. For example, the widget demonstration that comes with Tk works without having to make any changes. Because Tksh can interpret Tcl, it is able to work with the the large number of existing Tcl/Tk scripts. Thus, an application can be built with Tksh where the main functionality is written in ksh93, and widgets are used that have already been written in Tcl. The transition from Tcl/Tk to Tksh is therefore a smooth one. For example, if we were writing a Tksh script that required a dialog box to open file, we could take code directly from one of the many Tcl implementations rather than translate it or rewrite it from scratch.

Functions declared with `proc` will be parsed as Tcl source when they are subsequently called, even after `source` returns. For example, if we have the following script:

```
function foo
{
        X=37
        print "$(bar test)"
}

source <<'EOT'
proc bar {args} {
        global X
        set X [expr $X + 1]
        return "Proc bar: args: $args, X: $X"
}
EOT
```

A call to the function `foo` will result with the following output:

```
Proc bar: args: test, X: 38
```

Note that the Tcl procedure `bar` is able to use and modify the shell variable `X`. It is also possible for Tcl source to invoke ksh93 functions and built-ins.

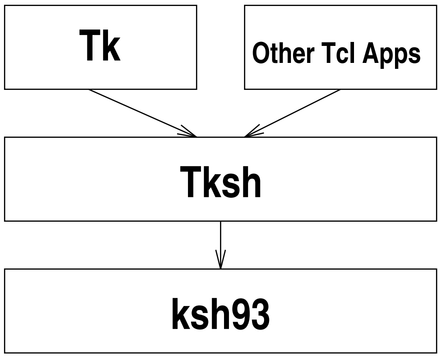


Figure B.3: Tksh Architecture

B.3 Tksh Architecture

Tksh is written in C and consists of around 5,000 lines of source code along with a few modules that have been taken directly from the source of Tcl. Most of the code is straightforward and maps the semantics of a Tcl library function to the corresponding ksh93 call (if such a function exists). In several cases, there is either no corresponding function in the API, or the semantics of the similar functions differ in a nontrivial way. This requires writing code to emulate the correct semantics. Examples of this are described throughout this section.

The parts of code that have been taken from Tcl without requiring modification include the hash table and dynamic string libraries, the Tcl parser, the implementation of most of the builtin commands (such as `proc` and `set`), and a few miscellaneous routines. While ksh93 has a hash table and string library, it is easier to use Tcl's routines for the moment because semantics of the libraries differ.

Variables and Traces

A trace is a function that is associated with an action performed on a particular Tcl variable. Traces can be set for three types of events: (1) when the value of the variable is read from, (2) when the value of the variable is written to, and (3) when the variable is unset (destroyed). ksh93 has an equivalent notion called *disciplines*. Discipline functions can be C functions or shell functions and can be stacked so that multiple discipline functions apply to a variable. At the shell level, functions whose names are in the format *variable_name.action* are discipline functions. The discipline actions `get`, `set`, and `unset` are permitted for all variables, and other discipline names can be defined using C code extensions.

Traces for variables are implemented on top of the discipline mechanism of ksh93. Although Tcl traces are similar to disciplines, there are a number of subtle differences in the mechanisms. For instance, when a Tcl trace is invoked on a variable, all other traces for that variable are turned off for the duration of the trace. ksh93, on the other hand, allows other disciplines to execute as long as they are further down on the discipline stack. Although it would have been possible to implement a separate tracing facility that was called from the `Tcl_GetVar` and `Tcl_SetVar` functions, using ksh93 disciplines is advantageous for several reasons reasons. If a variable is read from, written to or unset from a shell script, `Tcl_GetVar` and `Tcl_SetVar` are never called (and thus the traces wouldn't be). However, if the traces are installed as disciplines, they will be called in this case. Also, this implementation allows traces and disciplines to be interleaved on the same variable – a trace can be followed by a discipline which is followed by another trace.

Evaluation of Tcl Code

The code for Tcl is structured so well that it was possible to place Tcl's parser directly into Tksh without modification. The Tcl parser goes through a string and converts it into an argument list, performing any necessary variable or command substitutions on the way. Variable substitutions are done by using the library function `Tcl_GetVar`, which (as mentioned previously) has been rewritten to use the ksh93 API. Command substitutions are done by calling `Tcl_Eval` which recursively calls the parser.

The function `Tcl_Eval` calls the parser to generate the argument list, and then looks up the procedure to be invoked in the ksh93's built-in command table. If the command exists, it is executed with the argument list.

In order to interpret Tcl scripts in ksh, an internal flag has been added to Tksh that indicates which language is currently being interpreted. If we are interpreting ksh code, the `Tcl_Eval` and `Tcl_EvalFile` routines effectively hand off the string to be evaluated to the ksh93 equivalent, `sh_eval`. If we are interpreting Tcl code, the Tcl parsing routines are called instead, which generate an `argv` list to evaluated. Before the `argv` list can then be handed off to `sh_eval`, command names are mapped appropriately – Tcl command names such as `set` and `proc` are mapped to names that won't conflict with ksh93 (`tcl_set` and `tcl_proc`).

The Tksh built-in command `source` sets the aforementioned flag to indicate Tcl code is to be evaluated, and calls `Tcl_EvalFile` with the argument. When `source` has completed, the flag is restored to its original state.

Another modification has been made to the way the Tcl command `proc` behaves. Normally, the implementation of the `proc` command creates a Tcl command with the given name that corresponds to a C function named `InterpProc`. When `InterpProc` is called, it passes a field of its private data, which contains the string of the function body, to `Tcl_Eval`.

In Tksh, `InterpProc` first sets the flag indicating Tcl code is to be evaluated before calling `Tcl_Eval` on the function body. This way, a Tcl function can be called at any time (in ksh or Tcl code), not just within the `source` command.

Dynamic Scope

One important difference between ksh93 and Tcl is that Tcl uses dynamic scoping, while ksh93 uses static scoping (Tcl uses dynamic scoping in order to access an array defined in an enclosing function; ksh93 uses reference variables to accomplish this). The Tcl commands `upvar` and `uplevel` make use of such dynamic scoping, and have been rewritten in Tksh. Although the stack of enclosing scopes is inaccessible at the scripting level of ksh93, they are accessible through the C language API. Tksh uses these functions to achieve the correct behavior.

Files and Processes

Tcl has a set of commands that deal with files and processes. This part of Tcl has been rewritten in Tksh to make use of ksh93's handing of processes and files. In Tcl, a function called `Tcl_CreatePipeline` exists to start a set of processes and communicate with them (this library function is used by Tcl's `exec` command). Tksh implements this function by putting together an equivalent ksh command string from the arguments and evaluating it in the shell. For example, the Tcl command:

```
exec cat << "Hello" | tee foo >@stderr
```

would be translated to the following shell command:

```
cat <<HUP | tee foo 1>&2
Hello
HUP
```

In shell, only 10 files can be referred to at a time (by specifying a number from 0 to 9 in front of a redirection symbol). In Tcl, the number of files that can be specified at once is bound by the process limit and thus it is possible to specify more files at once in Tcl scripts. Tksh gets around this problem by mapping as many Tcl files to descriptors 0 through 9 as possible. If a file is not mapped, shell descriptors are moved around to map the file before the command is invoked, and restored afterwards. This implementation has the advantage that it allows files to be shared by ksh93 scripts and Tcl scripts, since ksh93's underlying file facilities are used for both.

Events

In Tcl/Tk, execution is driven by an event loop. Tk has a mechanism which is used to add events which will invoke a given procedure when the event occurs. ksh93, on the other hand, has it's own event mechanism that is part of the shell (the newest releases of Tcl/Tk have moved the event loop from Tk into Tcl).

The library for ksh93 contains a routine called `sh_waitnotify` which is used to specify a function to be called when the shell is waiting for either input or a child process to complete. The notify function should return when input is ready or when a signal is received (it must also return within timeout, which is specified). Tksh uses the `sh_waitnotify` mechanism to run the Tk event loop. It registers an event for the file descriptor that the shell is waiting on, as well as an event to occur after the timeout period. The function `Tk_DoOneEvent` is repeatedly called until such an event occurs. Signal handlers are also placed to trigger an event when a relevant signal occurs.

The event mechanism in ksh93 has a major advantage over Tcl's. If a command is executed, ksh93 will process events while waiting for the command to complete. Tcl, on the other hand, will freeze until the child process terminates. ksh93 is also able to parse its input as it is entered with this mechanism since it owns the event loop.

Quoting

The quoting constructs in Tcl are designed to be easily nested. For example, `[...]` notation is used for command substitution and {`...`} is used for literal quoting. KornShell behaves similarly for command substitution by introducing the `$(...)` syntax to replace the `` `...` `` syntax from Bourne shell.

A string is converted into a ksh93-style list element using the function `sh_fmtq`, which returns a shell-quoted version of its input string. A list is thus created by concatenating quoted versions of each element.

Regular Expressions

Tcl uses an extended form of regular expressions for pattern matching. Although shells traditionally provide notation that is far more restrictive than regular expressions, ksh93 also uses an extended form of pattern matching that is very similar to Tcl. The pattern matcher in ksh93 includes a way to negate expressions, which is not currently possible with Tcl (for example, it is possible to invoke the command `ls !(*.o)` to list all files that don't end in `.o`).

The ksh93 API has a routine which converts a regular expression into a shell pattern. Since shell patterns are more powerful than regular expressions, the translation from regular expressions is fairly simple. The Tcl regular expression functions are implemented by first translating regular expressions into shell patterns, and then calling the appropriate ksh93 library function.

Results

Commands in Tcl produce two results. First, there is an integer completion code that indicates success (with `TCL_OK`) or failure (with `TCL_ERROR`). Second, there is a field in the interpreter structure that points to a result string. The result string may either be the result of the successful command or the error message generated by an unsuccessful one. ksh93 is similar, but uses uses standard output and standard error to return strings. This model makes it possible for shell functions and built-ins to behave like external UNIX commands.

B.4 Summary

KornShell has many features that make it a suitable choice for writing scripts. It has functionality similar to both Tcl and Perl, good performance, and syntax that is less complex than Perl yet requires fewer levels of nested evaluation than Tcl does. It is also compatible with the Bourne Shell and conforms to POSIX and international standards. The creation of an interface to the internals of ksh93 which is equivalent to the interface to Tcl internals allows ksh93 to be used with existing Tcl applications. The most notable of these applications is Tk. In addition, since Tksh can interpret Tcl scripts, existing Tcl/Tk scripts can be used with Tksh.

Bibliography

[ACSW96] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. Tioga-2: A direct manipulation database visualization environment. In *Proceedings of the 12th International Conference on Data Engineering*, pages 208–217, Washington - Brussels - Tokyo, February 1996. IEEE Computer Society.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[Bae81] Ronald Baecker. Sorting out sorting, 30 minute color sound film. In *ACM SIGGRAPH Conference*, Dallas, TX, August 1981.

[Bae88] R. Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In *Proceedings of the 10th International Conference on Software Engineering*, pages 356–366, Singapore, April 1988. IEEE Computer Society Press.

[Bei90] Boris Beizer. *Software Testing Techniques, Second Edition*. Van Nostrand Reinhold, NY, 1990.

[BK95] Morris I. Bolsky and David G. Korn. *The New KornShell Command And Programming Language*. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, second edition, 1995.

[BMG+94] E. Buss, R. De Mori, W.M. Gentleman, J. Henshaw, J. Johnson, K. Kontogianis, E. Merlo, H.A. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S.R. Tilley, J. Troster, and K. Wong. Investigating Reverse Engineering Technologies for the CAS Program Understanding Project. *IBM Systems Journal*, 33(3):477–500, 1994.

[BML97] Naser S. Barghouti, John Mocenigo, and Wenke Lee. Grappa: A Graph Package in Java. In *Fifth International Symposium on Graph Drawing*, pages 336–343. Springer-Verlag, September 1997.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Englewood Cliffs, New Jersey: Prentice Hall, 1981.

[BP98] Ronald Baecker and Blaine Price. The early history of software visualization. In John Stasko, John Domingue, Marc Brown, and Blaine Price, editors, *Software Visualization*, chapter 2, pages 29–34. MIT Press, 1998.

[Bro88] M. H. Brown. Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5):14–38, May 1988.

[Bro91a] Marc H. Brown. An anthology of algorithm animations using zeus. 76b, Digital Equipment Corporation, Systems Research Centre, 30 August 1991. (time 59:00).

[Bro91b] Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *IEEE Workshop on Visual Languages*, pages 4–9, Kobe, Japan, 1991.

[BS84] Marc H. Brown and Robert Sedgwick. A system for algorithm animation. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 177–186, July 1984.

[BW83] Peter Bates and Jack C. Wileden. An approach to high-level debugging of distributed systems. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging,* in *SIGPLAN Notices*, 18(8):107–111, August 1983.

[BYJQ92] R. A. Baeza-Yates, L. Jara, and G. Quezada. Vcc: Automatic animations of c programs. In *Proceedings of Compugraphics*, pages 389–397, December 1992.

[CDN88] John P. Chin, Virginia A. Diehl, and Kent L. Norman. Development of an instrument measuring user satisfaction of the human-computer interface. In *Proceedings of ACM CHI'88 Conference on Human Factors in Computing Systems*, Interface Evaluations, pages 213–218, 1988.

[CFKW95] Yih-Farn Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A Graphical Navigator for Software and Document Repositories. In *International Conference on Software Maintenance*, pages 66–75, 1995.

[CGK97] Yih-Farn Chen, Emden Gansner, and Eleftherios Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 1997.

[Che76] P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Che95] Yih-Farn Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.

[CI90] Elliot H. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1), January 1990.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CK97] Yih-Farn Chen and Eleftherios Koutsofios. WebCiao: A Website Visualization and Tracking System. In *WebNet97*, 1997.

[CLVM98] J. C. Cunha, J. Lourenco, J. Vieira, and B. Moscao. A framework to support parallel and distributed debugging. *Lecture Notes in Computer Science*, 1401:708, 1998.

[CNR90] Yih-Farn Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[CRV94] Yih-Farn Chen, David Rosenblum, and Kiem-Phong Vo. TestTube: A System for Selective Regression Testing. In *The 16th Internation Conference on Software Engineering*, pages 211–220, 1994.

[CS95] Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets*. The Free Press, New York, N.Y., 1995.

[DB92] Christopher J. DiGiano and Ronald M. Baecker. Program auralization: Sound enhancements to the programming environment. In *Proceedings of Graphics Interface '92*, pages 44–52, May 1992.

[DBCK98] Fred Douglis, Thomas Ball, Yih-Farn Chen, and Eleftherios Koutsofios. The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web. *World Wide Web*, 1(1):27–44, 1998.

[Dev92] P. Devanbu. Genoa—a language and front-end independent source code analyzer generator. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 307–317, 1992.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[Dui87] R. A. Duisberg. Visual programming of program visualizations: A gestural interface for anumating algorithms. In *IEEE Workshop on Visual Languages*, pages 55–66, Washington, D.C., 1987.

[Eis97] Eisenstadt. My hairiest bug war stories. *CACM: Communications of the ACM*, 40, 1997.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, Menlo Park, CA, 1995.

[FJ98] Loe Feijs and Roel De Jong. 3D visualization of software architectures. *Communications of the ACM*, 41(12):73–78, December 1998.

[GH93] Michael Golan and David R. Hanson. DUEL—a very high-level debugging language. In *Proceedings of the Winter USENIX Technical Conference*, pages 107–117, San Diego, CA, January 1993.

[Gib94] W Wayt Gibbs. Trends in computing: Software's chronic crisis. *Scientific American*, 271(3):72–86, September 1994.

[Gla98] Robert L. Glass. *Software Runaways: Lessons Learned From Massive Software Project Failures.* Prentice Hall, Upper Saddle River, 1998.

[GR90] Charles F. Goldfarb and Yuri Rubinsky. *The SGML handbook*. Clarendon Press, Oxford, UK, 1990.

[Gra83] Wayne C. Gramlich. Debugging methodology (session summary). *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging,* in *SIGPLAN Notices*, 18(8):1–3, August 1983.

[Gra88] T.C. Nicholas Graham. Conceptual views of data structures as a programming aid. Technical Report 88-225, Queen's University at Kingston, Kingston, Canada, August 1988.

[Gra92] Judith Grass. Cdiff: A syntex directed differencer for c++ programs. In *Proceedings of the Usenix C++ Conference*, August 1992.

[GvN47] H. Goldstein and J. von Neumann. Planning and coding problems of an electronic computing instrument. In A. Taub, editor, *J. von Neumann Collected Works*, pages 80–151. Macmillian, 1947.

[Han76] David R. Hanson. Variable associations in SNOBOL4. *Software—Practice and Experience*, 6(2):245–254, April 1976.

[HB95] D. Hutchens and R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, 11:749–757, August 1995.

[Hen82] John L. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(4):323–344, July 1982.

[HJ91] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 125–138, Berkeley, CA, USA, January 1991. Usenix Association.

[HK97] David R. Hanson and Jeffrey L. Korn. A simple and extensible graphical debugger. In USENIX, editor, *1997 Annual Technical Conference, January 6–10, 1997. Anaheim, CA*, pages 173–184, Berkeley, CA, USA, January 1997. USENIX.

[HR96] David R. Hanson and Mukund Raghavachari. A machine-independent debugger. *Software—Practice and Experience*, 26(11):1277–1299, November 1996.

[HWF90] Robert Henry, Kenneth M. Whaley, and Bruce Forstall. The University of Washington illustrating compiler. *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 25(6):223–233, June 1990.

[ISO87] Information Processing — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1). International Organization for Standardization and International Electrotechnical Committee, 1987. International Standard 8824.

[Joh77] Mark Scott Johnson. The design of a high-level, language independent symbolic debugging system. In *Proceedings of the ACM Annual Conference*, pages 315–322, Seattle, WA, October 1977.

[Joh78] Mark Scott Johnson. *The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment*. PhD thesis, The University of British Columbia, August 1978.

[Joh81] Mark Scott Johnson. Dispel — a run-time debugging language. *Computer Languages*, 6(2):79–94, 1981.

[JW99] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *Proc. 21st Intl. Conf. Software Engineering*, May 1999.

[KA98] Jeffrey L. Korn and Andrew W. Appel. Traversal-based visualization of data structures. In *Proceedings IEEE Symposium on Information Visualization 1998*, pages 11–18. IEEE, 1998.

[KCK99] Jeffrey Korn, Yih-Farn Chen, and Eleftherios Koutsofios. Chava: Reverse engineering and tracking of java applets. In *WCRE'99 Proceedings*. IEEE, 1999.

[KH96] Adam Kolawa and Arthur Hicken. *Insure++: A Tool to Support Total Quality Software*. ParaSoft Corporation, Monrova, CA, 1996.

[KNK96] David G. Korn, Charles J. Northrup, and Jeffery Korn. The new Korn shell. *Linux Journal*, 27:??–??, July 1996.

[Knu88] Donald E. Knuth. The errors of TEX. Technical Report STAN-CS-88-1223, Stanford University, Department of Computer Science, September 1988. See [Knu89].

[Knu89] Donald E. Knuth. The errors of TEX. *Software—Practice and Experience*, 19(7):607–685, July 1989. This is an updated version of [Knu88]. Reprinted with additions and corrections in [Knu92, 243–339].

[Knu92] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992. Distributed by the University of Chicago Press.

[Kor96] Jeffrey L. Korn. Tksh: A Tcl library for KornShell. In *Proceedings of the USENIX Tcl/Tk Workshop*, pages 149–159, Monterey, CA, July 1996.

[Lie97] Derek Lieber. The jikes debugger. Jikes is available for download at `http://www.alphaworks.ibm.com/formula/jikesdebugger`, 1997.

[LRB+97] Miron Livny, Raghu Ramakrishnan, Kevin S. Beyer, Guangshun Chen, Donko Donjerkovic, Shilpa Lawande, Jussi Myllymaki, and R. Kent Wenger. DEVise: Integrated querying and visualization of large datasets. In Joan Peckham, editor, *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 301–312, Tucson, Arizona, 13–15 June 1997.

[McC97] C. McCreary. The vgj graph drawing tool. `http://www.eng.auburn.edu/department/cse/research/graph_drawing/vgj.htm%l`, 1997.

[Mic98] Sun Microsystems. Java foundation classes. `http://java.sun.com/products/jfc`, 1998.

[Mil56] GA Miller. The magical number seven, plus or minus two: Some limtis on our capacity for processing information. *Psycological Review*, 63:81–97, 1956.

[MM91] Barton Miller and Charles McDowell, editors. *ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991. Available as *SIGPLAN Notices* 26(12), December, 1991.

[MMR+98] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Sixth International Workshop on Program Comprehension*, June 1998.

[Moh88] T. G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[MOTU93] H. Müller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[MS94] Sougata Mukherjea and John T. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger. *ACM Transactions on Computer-Human Interaction*, 1(3):215–244, 1994.

[Mye83] B. A. Myers. Incense: A system for displaying data structures. *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):115–125, July 1983.

[NB94] M. A. Najork and M. H. Brown. A library for visualizing combinatorial structures. In R. Daniel Bergeron and Arie E. Kaufman, editors, *Proceedings of the Conference on Visualization*, pages 164–171, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[NK94] Stephen C. North and Eleftherios Koutsofios. Applications of graph visualization. In *Proceedings of Graphics Interface '94*, pages 235–245, Banff, Alberta, Canada, May 1994. Canadian Information Processing Society.

[Oas99] The SGML/XML Web Page. `http://www.oasis-open.org/cover/xml.html`, 1999.

[OCHW91] Ronald A. Olsson, Richard H. Crawford, W. Wilson Ho, and Christopher E. Wee. Sequential debugging at a high level of abstraction. *IEEE Software*, 8(3):27–35, May 1991.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.

[PLVW98] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–235, Santa Fe, NM, April 1998. USENIX.

[PP94] S. Paul and A. Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*, 20(3):463–475, June 1994.

[RC89] G.-C. Roman and K. C. Cox. A declarative approach to visualizing concurrent computations. *Computer*, 22(10):25–36, October 1989.

[RCCC99] Herman Rao, Yih-Farn Chen, Ming-Feng Chen, and Josie Cheng. iproxy: An agent-based middleware. In *Poster Proceedings of the Eighth World Wide Web Conference*, May 1999.

[RCWP92] G.-C. Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun. Pavane: a system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(2):161–193, 1992.

[Ref98] The Java Reflection API. `http://www.javasoft.com/products`, 1998.

[RH92] Norman Ramsey and David R. Hanson. A retargetable debugger. *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 27(7):22–31, July 1992.

[RKK98] D. Rayside, S. Kerr, and K. Kontogiannis. Change and adaptive maintenance detection in java software systems. In *Proc. Fifth Working Conference on Reverse Engineering*, October 1998.

[RMD89] Steven P. Reiss, Scott Meyers, and Carolyn Duby. Using GELO to visualize software systems. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Tool Kits, pages 149–157, 1989.

[Rus92] Mark Russell. `ups`. `ups` is a graphical, source-level debugger for C. It is available for anonymous `ftp` from host `unix.hensa.ac.uk` (129.12.21.7), in `/misc/unix/ups/ups-2.45.tar.Z`. The author's email address is `mtr@ukc.ac.uk`., 1992.

[SB95] David Sharon and Rodney Bell. Tools that Bind: Creating Integrated Environments. *IEEE Software*, 12(2):76–85, March 1995.

[Sch91] R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. 13th Intl. Conf. Software Engineering*, May 1991.

[SDBP98] John Stasko, John Domingue, Marc Brown, and Blaine Price. *Software Visualization*. MIT Press, Cambridge, 1998.

[SGI95] SGI Technical Publications, Mountain View, CA. *Developer Magic: ProDev WorkShop Overview*, 1995.

[SI91] Takao Shimomura and Sadahiro Isoda. Linked-list visualization for debugging. *IEEE Software*, 8(3):44–51, May 1991.

[SP91] Richard M. Stallman and Roland H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, July 1991.

[Sta88] John Stasko. The TANGO algorithm animation system. Technical Report CS-88-20, Brown University, 1988.

[SvG98] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of java software. In *Proc. Foundation of Software Engineering*, November 1998.

[SZ90] Shasha and Zhang. Fast algorithms for the unit cost editing distance between trees. *ALGORITHMS: Journal of Algorithms*, 11, 1990.

[TFCB90] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.

[Tho89] Ian Thomas. PCTE Interfaces: Supporting Tools in Software-Engineering Environments. *IEEE Software*, 6(6):15–23, November 1989.

[Tol92] Andrew P. Tolmach. *Debugging Standard ML*. PhD thesis, Department of Computer Science, Princeton University, Princeton, NJ, October 1992.

[WAKS97] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 213–228, Berkeley, October 15–17 1997. USENIX Association.

[Win94] Phil Winterbottom. Acid: A debugger built from a language. In *Proceedings of the Winter USENIX Technical Conference*, pages 211–222, San Francisco, CA, January 1994.

[Zel73] M. V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9):566–566, September 1973.

[ZL96] Andreas Zeller and Dorothea Lütkehaus. DDD—A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, January 1996.