HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java

Jaishankar Sundararaman
Dep. of Computer Science
Virginia Tech
jaisunda@cs.vt.edu

Godmar Back
Dep. of Computer Science
Virginia Tech
gback@cs.vt.edu

Abstract

Data structure visualization can increase program understanding in an educational context and help with visual debugging. Existing data structure visualization tools are limited in interactivity, providing mostly static views; flexibility, by restricting the layout strategies users can apply; scope, by focusing on only a single language; and fidelity, by abstracting away the actual runtime layout and size of a program's data. This paper presents the design and implementation of HDPV, a system for interactive, faithful, in-vivo runtime state visualization for native C/C++ programs and Java programs. We discuss how HDPV can be used for a number of use cases ranging from understanding simple, recursive programs, to understanding the visual effect of programming errors such as buffer overflows.

CR Categories: K.3.2 [Computing Milieux]: Computers and Education—Computer and Information Science Education; E.2 [Data]: Data Storage Representations

Keywords: Software Visualization, Object Viewer

1 Introduction

Tools that can visualize a program's runtime state have been shown to be highly effective in program understanding and learning [Jain et al. 2006]. They have been used by educators to create visual algorithm animations from an algorithm's implementation, they can be used by students to understand the behavior of a program they are writing, or by practitioners to find errors or anomalies within the data structures their programs create. These tools are particularly effective for commonly used procedural and object-oriented languages whose runtime state is composed of per-thread automatic variables, global variables, and dynamically allocated heap objects.

Yet, existing tools such as jGRASP [Hendrix et al. 2007] or Jeliot 3 [Moreno et al. 2004] may fail to fully realize their potential impact because they suffer from a number of significant limitations.

First, and most importantly, these tools support only limited interaction with the displayed state. Users may be unable to change the layout of a visualization or its size, or cannot zoom into areas of interest or exclude irrevelant areas from the visualization. Such lack of interactivity severely limits the use of these tools for visual debugging. Furthermore, the limited size of most display devices limits scalability if users cannot focus in on areas of interest.

Second, few available tools support the user-directed application of different layout strategies to different subsets of the program state. For instance, if the program's heap contains both a tree and a list data structure, it should be possible to lay out the tree separately from the list.

Third, most tools do not visualize the state of a program *in-vivo*. Instead, an *in-vitro* behavior is being visualized. In such an artificial environment, important facets may be missed during visualization. For instance, an array overflow bug may corrupt variables that lie adjacent in memory during the actual execution, but the visualization tool may not be able to visualize this fact because it lacks information about the specific runtime layout that is used during execution.

Fourth, most tools are tied to a single language, often to Java. During the past years, Java has been adopted as the first programming language taught in many CS undergraduate programs. This trend has had a tremendous positive impact by allowing students to design data structures and algorithms without having to worry about low-level details such as pointers and memory management. However, shielding students from these details in introductory courses has led to well-documented difficulties in later systems and architecture courses that use C or C++, and has led to anecdotal evidence of employer dissatisfaction [Dewar and Schonberg 2008]. In the authors' experience teaching an undergraduate course in operating systems, students have difficulty grasping such concepts as how a thread's state can be saved on that thread's stack during a context switch. These difficulties could be reduced if a visualization tool applied the same techniques to either language. For example, students could see how Java's references correspond to C's pointers, how static fields correspond to global variables, and how C's `struct` and Java's `class` are similar.

To address these shortcomings, we have developed HDPV, an application for faithful 2D interactive, graph-based program state visualization. HDPV consists of language-dependent monitors that create and relay a stream of events about a program's in-vivo execution to a language-independent visualizer which displays the program's state to the user. A user can interact with HDPV to focus in on areas of interest, to apply different layout strategies to parts of the program state, or to rearrange its state for better inspection. Created state displays can be saved and restored. To achieve language-independent visualization, HDPV uses a canonical state model that closely resembles the ABI targeted by compilers such as gcc or g++. We have implemented monitors for compiled C/C++ code and for Java bytecode.

This paper describes HDPV's overall architecture, provides the design rationale for our canonical state representation, and describes the implementation strategy and challenges used for the monitors. We present the design and describe the implementation of the visualizer and provide a cognitive walkthrough through a number of example use cases.

2 Architecture

Figure 1 gives an overview of our architecture. Language-dependent monitors (for compiled C/C++ code and Java bytecode) use binary instrumentation to trace changes to a program's state. These state changes are expressed as events in a canonical machine

SOFTVIS 2008, Herrsching am Ammersee, Germany, September 16–17, 2008.

model. Events are encoded using a dialect of XML which we designed. The visualizer receives a stream of such events, and creates and maintains a "shadow state" model of the monitored program's state.

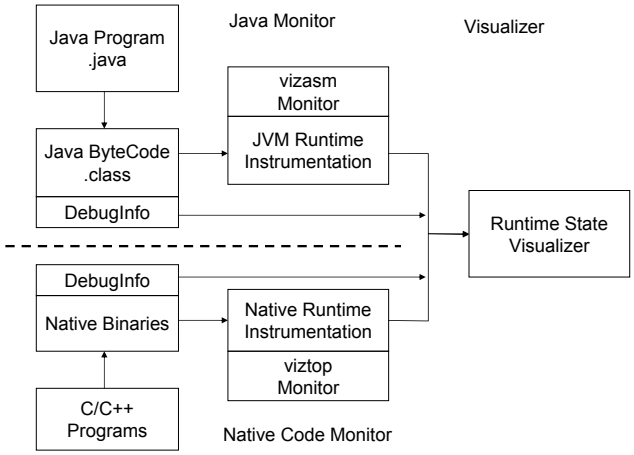


Figure 1: *An overview of the HDPV system.*

2.1 Canonical State Model

To accomplish our goal of providing a faithful visualization that reflects a program's data at run time, yet also provide the ability to display pointer-relationships between data objects, we needed to define a state model that reconciles the view of memory as a one-dimensional array of bytes (or octets) with that of a graph of objects that is suitable to a two-dimensional layout.

Our approach splits memory into intervals called *blocks*. These blocks form the nodes of a state graph. Nodes are typed, and one or more type definitions may be associated with each node. Multiple type definitions are required to support languages that allow pointers to an object to be cast to a different type—for instance, new objects in C are initially known to be of (C type) *char* *, and take on a more specific type only after being assigned to a pointer that is declared to be of that specific type (e.g., *struct S * p = malloc(sizeof S)*). Memory blocks are used to capture dynamically allocated objects, global variables, and activation records (stack frames) of currently executing functions. Edges connect memory blocks and represent points-to relationships between objects. Edges are directed and labeled with two offsets: a source offset represents the location within the source memory block into which a pointer is stored, and target offset represents an offset within the pointed-to memory block.

We defined a simple type system to express memory blocks' types. This type system includes primitive types modeled after C99's <stdint> header, such as signed and unsigned single and multiple byte integer types and Unicode characters. Type constructors can be used to define derived types: we support pointer types, array types, and record ("struct") types. As in C++ or Java, components of a record type are called fields. Fields may have hints associated with them: for instance, fields within an activation record that represent function arguments are labeled differently from fields that represent local variables.

2.2 Instrumentation Trace Protocol

Monitor and visualizer communicate via a stream-based protocol which transmits messages that describe relevant program events. Currently, this communication is unidirectional from the monitor to the visualizer. Our simple protocol includes six messages.

Typedef. A typedef message defines a record type and assigns a name to it. Each record type is represented as an ordered sequence of typed fields. The monitor can extract such type information either from debugging information in the executable (if available) or from runtime information (e.g., using Java reflection), or both. The visualizer will store all announced typedefs in a map indexed by name and may later use them to render the contents of a memory block using a particular type definition.

Memalloc. A memalloc message announces the creation of a new memory block, which includes an address and size. Typically, a monitor will instrument memory allocation routines to learn of allocation events. Additionally, the monitor may provide a hint that informs the visualizer of the allocated object's kind, allowing the visualizer to apply different layout strategies to stack, heap, or global objects. Upon receipt of a memalloc, the visualizer will add a new node to the state graph it maintains.

Memfree. A memfree message announces that an object has been destroyed. For instance, when a function returns or throws an exception, its activation record is destroyed. The visualizer will remove the node from the state graph along with all incoming and outgoing edges.

Putfield. Putfield messages announce updates to memory blocks. A putfield message consists of the address of the block being updated, the offset within the block, and the value being written, along with its associated type. A monitor will instrument assignments to local variables, object fields, global variables, and array elements to generate putfield messages. In response to a putfield event, the visualizer must update the base object's visual state, and may need to add edges to the state graph.

Location. Monitors send a location message to announce that execution has progressed to a particular source file and line number, allowing the visualizer to focus the user's attentions to that line.

Showas. A showas message, which takes as argument a memory block, is sent as a hint by the monitor to the visualizer to signal that a particular block should be displayed using a specific type, if multiple types are associated with a block. For instance, if a local variables goes into or out of scope inside a function, the monitor may signal this fact by sending a typedef message, and directing the visualizer to display the current activation record using this type definition.

3 Monitor Implementation

Program monitoring can be implemented using a number of methods: by using custom interpreters or compilers, using source-to-source transformation, using executable rewriting, or using debugging interfaces. Interestingly, closer examination reveals that none of these approaches supports our goal of providing faithful, in-vivo program monitoring: either because the method requires source code, or because it changes relevant aspects of the program's runtime state, such as memory addresses. We have found existing debugging interfaces such as JPDA to be unsuitable because they do not support notification for all relevant events that update state (e.g., JPDA does not support notifications when an array's content is being updated), thus requiring extensive polling. Therefore, we implemented our monitors using binary runtime instrumentation

methods that exactly reproduce the state the program would have if run without the monitor.

3.1 Native Code (viztop)

For compiled executables, we built a monitor using a layered architecture consisting of three components. At the bottom layer, we use Pin [Luk et al. 2005], a toolkit that allows the creation of low-level binary instrumentation tools for ARM, Itanium, and x86-compatible processors. We refer to Pin as "low-level" because it requires a tool writer to think in terms of machine instructions and registers. We developed and use Top [Gopal 2006] on top of Pin. Top is a C++ class library that facilitates the creation of higher-level instrumentation applications such as the viztop monitor. Top provides an event-based framework that allows clients such as viztop to register for such events as function entry or exit, memory allocations, and accesses to allocated objects. Based on these events, it creates the state-related events described in Section 2.2. Most of the events can be straightforwardly created from the events provided by Top, but some required additional work. In particular, in order to correctly compute putfield events for variable assignments and object or array updates, viztop needs to maintain an interval tree data structure that can map an address to its containing object.

To announce the allocation of stack frames as activation records, and to map memory accesses on a thread's stack to accesses to local variables, viztop must compute the layout of each stack frame, and announce this layout as a type definition to the visualizer. We adopted a substantially extended version of the Fjalar library, which is part of the Daikon system [Ernst et al. 2007], to retrieve such type and local variable information from the DWARF debugging information section of native executables, if available. To provide a faithful image of a process's stack, we include elements such as the memory location in which the return address left by a CALL instruction is being stored.

Issues and Limitations. Our current implementation does not trace register values, although we plan on extending it to do so to provide higher levels of resolution. From the user's point of view, data that is held solely in registers will not appear in the visualization until it is written to memory. From a practical point of view, our approach works best if a compiler produces code that allocates all local variables on the stack and issues a load/store instruction before every operation—which is the behavior of most compilers, including gcc, when producing unoptimized code. We also currently do not report and relay temporaries, such as the arguments being passed to a function call, while they are being computed. The user will see such values appear in the visualization when the function being called is entered.

3.2 Java Code (vizasm)

Our Java monitor (vizasm) uses dynamic bytecode rewriting to inject code that reports on a program's execution. vizasm can be used as a replacement system class loader, which instruments loaded Java classes' bytecode during the JVM's class loading process. It can also be used as a Transformer with Java 6's instrumentation framework, allowing it to redefine already loaded classes. We used the ASM library [Bruneton et al. 2002] to parse and instrument class files.

The runtime state of the Java virtual machine, which Java bytecode targets, consists of a dynamic heap of objects, global variables (in the form of static class variables), and per-thread stacks of activation records. Each activation record contains local registers, representing local variables, and a stack area that is used by bytecode for intermediate computation results. Our monitor intercepts all updates to objects' instance fields, classes' static fields, local registers, and arrays, and reports putfield events to the visualizer. Method entry and exits are reported as allocation and deallocation events of their respective stackframes. Each method is wrapped in a try/catch clause that catches all exceptions and errors thrown and issues the required memfree events to allow the visualizer to unwrap the stack. If the bytecode contains debugging information about local variables, vizasm will compute a type definition for a stack frame layout and issue appropriate showas messages when a particular local variable goes into scope.

Unlike the native code monitor, vizasm produces state updates in the context of a type-safe virtual machine, which does not expose the actual memory layout or addresses. For this reason, we assign addresses to Java objects for the purposes of visualization. These addresses are designed to mimic the addresses and layout that would be encountered had the Java code been compiled with a Java ahead-of-time compiler such as gcj. For the runtime stack, we chose a downward growing stack and assumed Pascal parameter passing style.

Issues and Limitations. Similar to our native monitor, our current implementation does not trace the values of the Java stack area, so temporary values are not visible to the user until they are assigned to a local variable. Since the Java stack area is also used to compute method call arguments, the computed arguments will not be visible until the method called is being entered. Our current implementation of vizasm cannot currently instrument Java classes on which it itself depends, such as java.lang.System. As a consequence, our implementation may encounter references to object whose creation it did not observe; in this case, it must use reflection to recursively explore those objects, and announce their type and existence to the visualizer. Finally, because the code instrumented by vizasm must pass bytecode verification, it cannot operate on new objects until after the superclass constructor has been invoked.

4 Visualizer

We implemented our visualizer using the prefuse toolkit [Heer et al. 2005]. Prefuse is an extensible software framework for creating interactive information visualization applications. It supports the interactive visualization of tables, graphs, and trees, implementing necessary components such as layout algorithms and input controls. It is written in Java in an object-oriented style that makes heavy use of design patterns, allowing users to adapt and extend its components via subclassing and substitution. Since our canonical state model represents a graph data structure at its core, we used prefuse's graph visualization components, which we heavily customized. We implemented specific renderers, specific layouts, and designed an interaction strategy.

4.1 Rendering Memory Blocks

Memory blocks are rendered either horizontally or vertically as record diagrams. We display the current type definition used for rendering, the names of each field, and the value in a format corresponding to that field. A tooltip displays the currently chosen type definition. Figure 2 shows an example object of type `java.lang.String`. It has four fields: a reference field 'value' that points to a Java 'char[]' array, and three 32-bit integers 'offset,' 'count,' and 'hash.' In order to convey information about the relative and absolute size of fields to the user, we decided on drawing all objects to scale with respect to the memory they occupy. For large blocks such as arrays, the user has the option of seeing the entire object, or only a truncated version of it. The user has the option of choosing between a Java-style view and a C/C++-style view, which respects the type naming conventions of the chosen language.

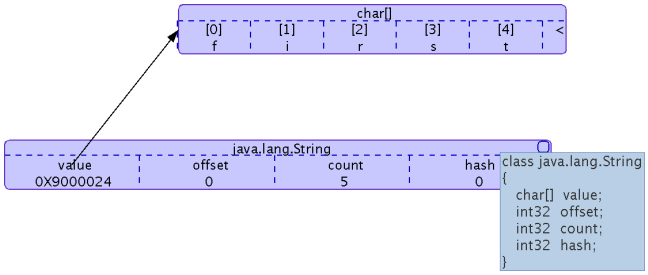


Figure 2: *Rendering of a* `java.lang.String` *instance. A tooltip displays the type definition. A user can collapse any object by clicking on the ◯ located at the top right corner of an object.*

4.2 Layout Strategy

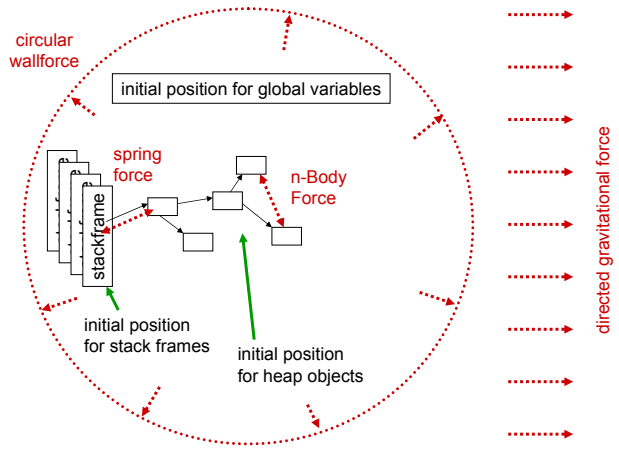
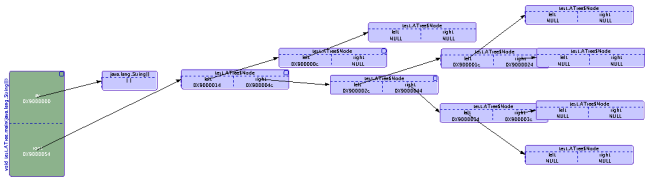


Figure 3: *Initial positions and forces in force-directed layout.*

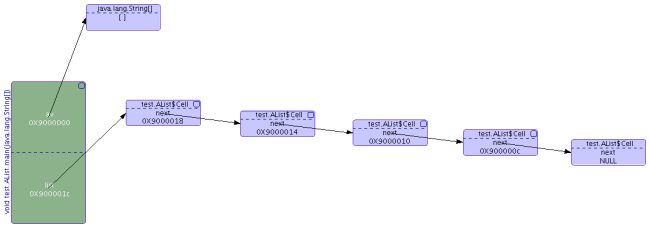
We designed a layout strategy that both provides reasonable behavior for small heaps in the absence of any user interaction and that performs well also when the user interactively assigns positions to some memory blocks. Figure 3 shows our strategy. In this layout, nodes may be either fixed or floating. A fixed node's position will not change when a layout is recomputed. When a stack node or global variable is allocated and added to the visualization, its position is initially fixed. The coordinates are computed from the underlying memory block's address. We lay out globals left to right in increasing address order. Stack nodes, which correspond to activation records, are laid out top down. Alternatively, the user can choose a collapsed, or stacked, layout of the currently active frames, as shown in the figure. Heap objects, on the other hand, are placed initially in the center of the layout; they are placed in the floating state.

The position of floating nodes is computed via a simple force simulation consisting of multiple forces. A repelling n-body force acts on all floating nodes. In addition, a spring force acts between nodes that are connected via an edge. A drag force acts in the direction opposite of that towards which the node currently moves, which leads to smoother animated behavior. In addition to these three forces, which are provided by prefuse's default force layout implementation, we added two additional forces: a circular wall force that acts radially from the insertion point, and a directional gravitational force that acts horizontally to the right. A preferences panel allows the user to adjust the strength of all forces if desired.



```
public class ATree {
    static class Node {
        Node left, right;
        Node(Node left, Node right) {
            this.right = right;
            this.left = left;
        }
        Node() { this(null, null); }
    }
    public static void main(String []av) {
        Node root =
            new Node(
                new Node(
                    new Node(),
                    null),
                new Node(
                    new Node(
                        new Node(),
                        new Node()),
                    new Node(
                        new Node(),
                        new Node()))));
    }
}
```

Figure 4: *Default layout for simple tree. This layout was produced via force simulation from the code shown above, without user interaction. The stack frame on the left contains parameter* `av` *and local variable* `root`*.*



```
public class AList {
    static class Cell {
        Cell next;
        Cell(Cell next) { this.next = next; }
        Cell()          { this(null);       }
    }
    public static void main(String []av) {
        Cell list =
            new Cell(
                new Cell(
                    new Cell(
                        new Cell(
                            new Cell()))));
    }
}
```

Figure 5: *Default layout for a simple linked list prior to user interaction. The program that produced this layout is shown as well.*

Combined, these initial positions and initial states lead to reasonable default behavior. For instance, Figures 4 and 5 show the layouts computed for a toy tree and list implementation.

4.3 Interaction Strategy

Prefuse provides a number of controls that allow user input via the mouse. These include controls for panning, zooming, and resizing the display. A drag control supports dragging of individual nodes as well as of subtrees of nodes, which are computed from the graph's underlying spanning tree. If a floating node is dragged, it will become fixed in the new position. We implemented additional controls to allow a user to fix and unfix a node's position as well as controls that allow a user to highlight, expand or collapse all nodes within the subtree emanating from a node. To highlight, the user simply has to hover over a node. A user can choose to apply a node-link tree layout diagram [Buchheim et al. 2002] to any particular subtree. HDPV keeps track of which nodes are currently subject to a node-link tree layout. When a new object is created and added to a tree, the layout is rerun, moving the node in the position it should have according to the tree layout. There may be multiple, independent node-link tree layouts in effect for different areas of the program's state graph. A preference panel allows the user to adjust breadth and depth spacings for each node-link tree layout individually.

To control the overall progress of the visualization, we provide buttons that allow a user to skip forward to the next visual change in the state graph, or to skip forward until execution returns from the current function or method, or an exception is thrown. If source code is available, we display the code using a syntax-coloring editor, and highlight the current source code location. In that case, a button is provided that allows the user to skip forward to the next reported source code location. Lastly, we implemented an animation mode in which the visualization processes events in fixed time intervals. The user can start and stop the animation and change its speed. Users can save a given layout to disk and restore it from there.

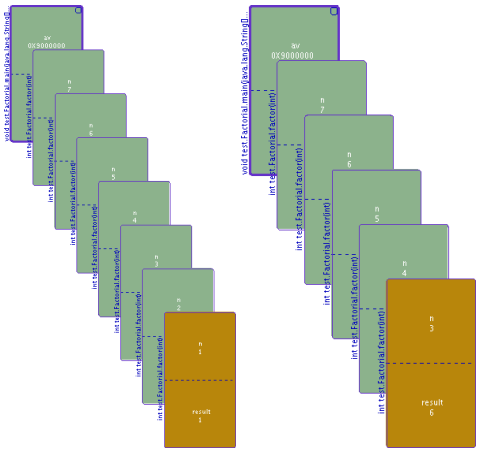
Issues and Limitations. Our implementation exhibits a number of limitations, which we are currently addressing. First, it lacks a "rewind" functionality. Second, we have not prepared a sensible initial layout strategy for multi-threaded programs. Third, the node-link tree layout is unaware of "null" fields, resulting in a sometimes non-intuitive layout of trees.

5 Example Use Cases

To evaluate the usability of our systems, we present a brief cognitive walkthrough through a number of hypothetical tasks for which a user may wish to use our system. These tasks may be encountered by students in introductory programming classes when doing assignments or debugging their code, or they may be tasks an instructor may face when creating illustrations of concepts. From our teaching experience, we believe that a faithful visualization of runtime state could become an important teaching tool in an introductory compilers or operating systems class. For instance, students implementing a compiler must understand how to lay out stack frames and objects by assigning offsets to local variables and fields, respectively.

5.1 Understanding Recursion

We studied how HDPV could contribute to understanding how local variables contained in activation records are being used to store the state of a recursive program. A classic example of a recursive program is the computation of the factorial of a number, as shown in Figure 6. In this example, as the recursion progresses, new stack frames are being added. Once the end condition to stop the recursion is reached, the stack is gradually unwound and computed results appear in the stack frame of the calling method. Failure to include a proper end condition would be visualized by an indefinitely growing number of activation records.



```
public class Factorial {
    static int factor(int n) {
        int result = 1;
        if (n > 1)
            result = n * factor(n-1);
        return result;
    }

    public static void main(String []av) {
        int f7 = factor(7);
        System.out.println(f7);
    }
}
```

Figure 6: *A simple factorial routine, as shown in the code below. The left shows the visualization after 7 calls to 'factor' have been made, the snapshot on the right shows the state after 3 of these 7 calls have returned. Note that the stack frame of method 'main' is collapsed, hiding the* `String [] av` *array.*

5.2 Understanding Pointers

Understanding Call-By-Reference. In C, call-by-reference semantics is implemented using pointers. Figure 7 shows that HDPV can accurately produce this relationship.

Understanding 'this'. 'this' is typically implemented as an additional, invisible argument to each method call, which is how we visualize it. For instance, consider the following extension to the program shown in Figure 4:

```
public class ATree {
    static class Node {
        ...
        void traverse() {
            if (this.left)
                this.left.traverse();
            if (this.right)
                this.right.traverse();
        }
    }
}
```

In this traversal algorithm, the current traversal position in the tree is kept track of by the 'this' pointer pointing to the node currently

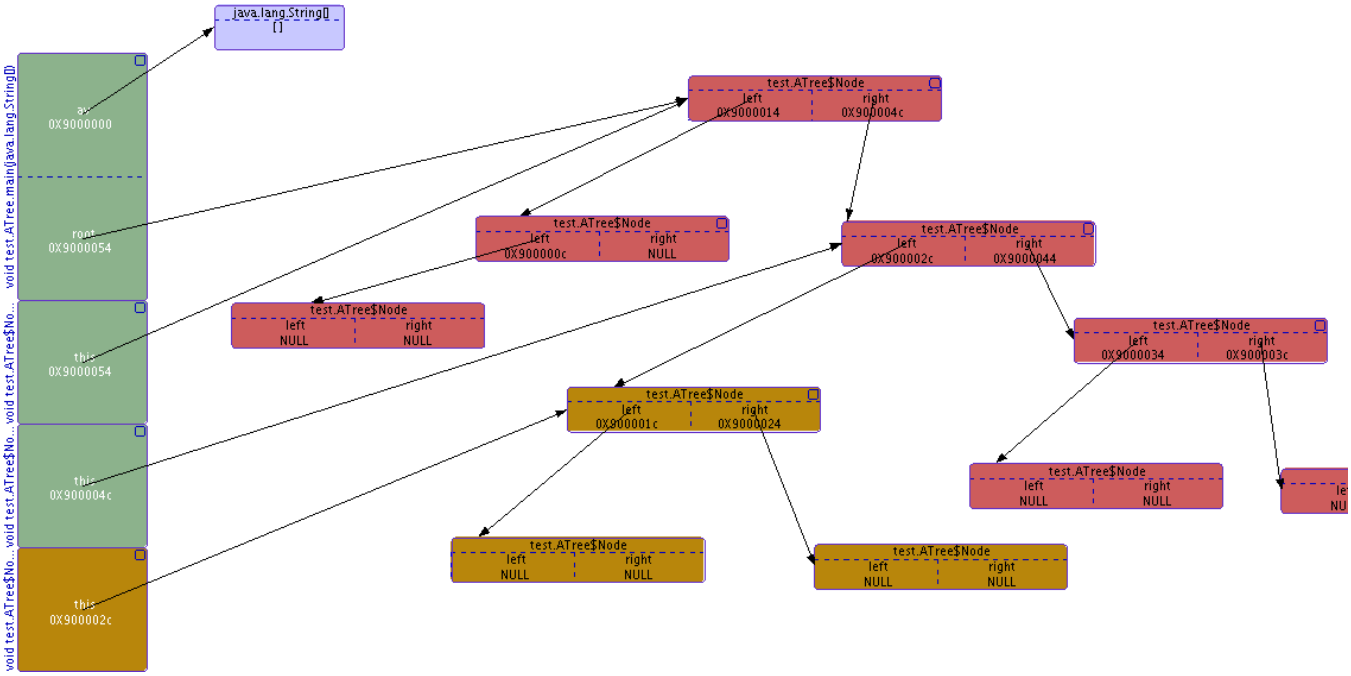
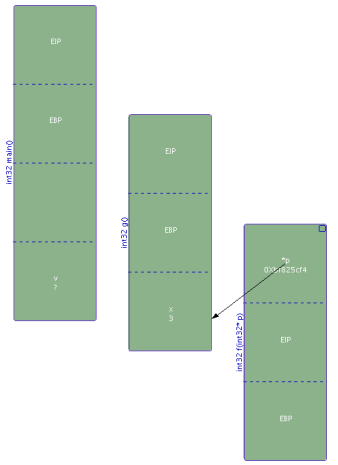


Figure 8: *Understanding 'this.' During a recursively implemented tree traversal, the 'this' pointer refers to the current node. As the user hovers over stack frames (shown left), the subtree reachable from the 'this' pointer in this stack frame is highlighted.*




```
int f(int *p) {
    *p = 3;
}

int g() {
    int x;
    f(&x);
    return x;
}

int main()
{
    int v = g();
    printf("%d\n", v);
}
```

Figure 7: *A simple example to visualize the concept of passing arguments by reference. Function f() accepts a single argument, which points to a location in g's activation record.*

being traversed. A user could use our system to visualize this fact. After creating the tree 'root,' the user could activate a vertical layout for the subtree emanating from 'root'. For the display of the runtime stack, the user may choose a vertical (e.g. non-collapsed view). When subsequently following the updates made by the program, the user will see how new stack frames are added, each consisting of a single 'this' pointer pointing to the root of the subtree that is currently being traversed. By using the hover control, the user can go through the stack, at which point all nodes within the subtree being traversed at this particular point in the activation stack will be highlighted. Figure 8 shows a snapshot of what a user sees during this interaction.

5.3 Making Sense of Larger Data Structures

If we extend the program shown in Figure 5 as follows:

```
static class Cell {
    ATree.Node tree = new ATree.Node(
                new ATree.Node(
                    new ATree.Node(),
                    null),
                new ATree.Node(
                    new ATree.Node(
                        new ATree.Node(),
                        new ATree.Node()),
                    new ATree.Node(
                        new ATree.Node(),
                        new ATree.Node()))));
    ...
}
```

we obtain a heap with 55 objects. This heap represents a linked list of trees. Initially, these nodes will appear as a crowded accumulation of nodes. However, by using the subtree drag control and applying tree layouts in a trial and error fashion, a user can quickly arrange those objects as shown in Figure 9.

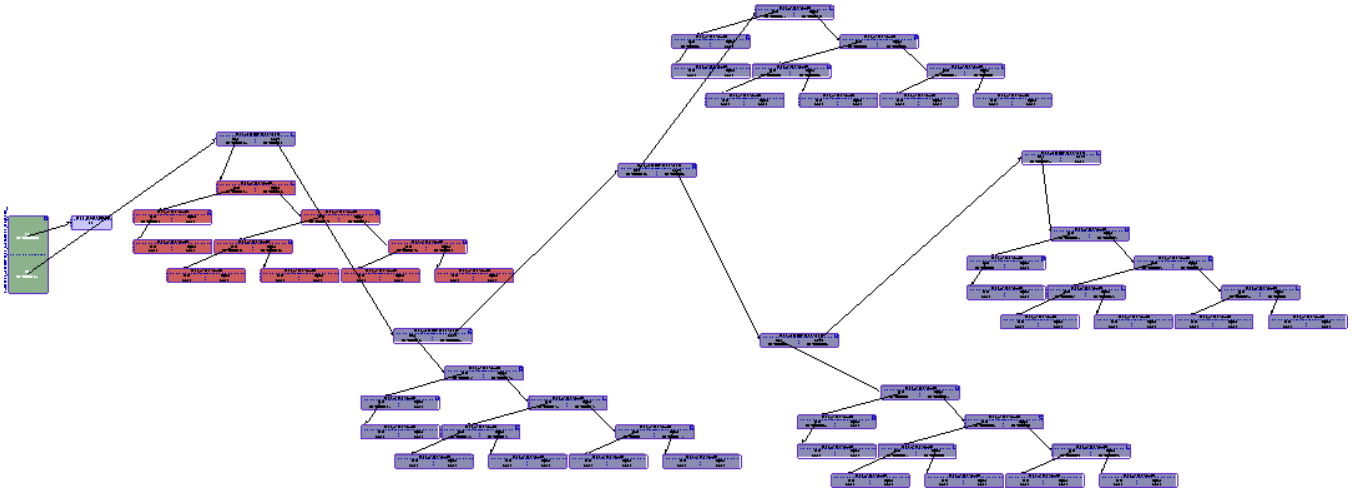


Figure 9: *A list of trees. This arrangement was created interactively.*

5.4 Programming Errors

Certain programming errors can only be understood by realizing how they impact a program's runtime state. Two examples include buffer overflow errors and memory leaks.

5.4.1 Buffer Overflows

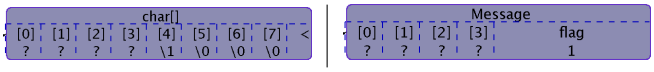


Figure 10: *An object of type 'Message', shown using two type definitions. This snapshot shows the state of the object before a buffer overrun error. Uninitialized memory (such as the memory returned from malloc()) is shown using a ?. The user can switch between the left and right view by double-clicking on the object.*

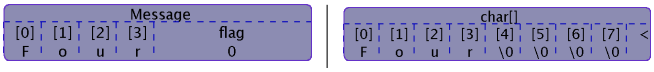


Figure 11: *The buffer overflow overwrote the least significant byte of 'flag,' changing its value to zero. A user can observe the overlap between flag and the zero sentinel placed by strcpy() when copying the string "Four" into a buffer that was too small.*

Consider the following buffer overflow:

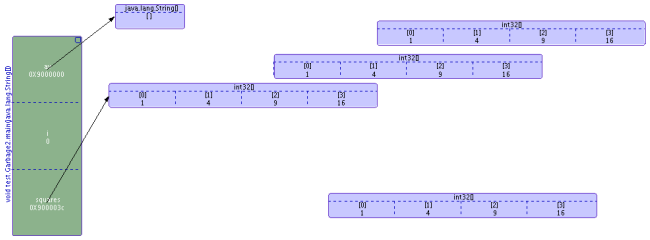
```
class Message {
   char buf[4];
   int  flag;
}

const char * msg4 = "Four";

Message * m = (Message*)malloc(sizeof *m);
m->flag = 1;
strcpy(m->buf, msg4);
```

In this example, the programmer failed to account for the trailing zero sentinel that terminates a C string. As a result, the first byte following buf[3] will be set to 0, which happens to be part of the 'flag' field. As a result, the value of flag will be changed to 0 (assuming a little-endian machine such as the x86). Figures 10 and 11 show a snapshot of how HDPV visualizes the object in question before and after the call to strcpy().

5.4.2 Memory Leaks, Churn, and Anomalies



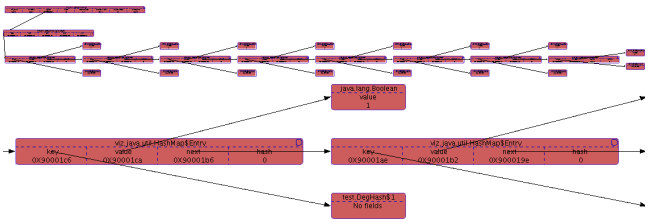
```
for (int i = 0; i < 4; i++) {
    int [] squares = new int[] { 1, 4, 9, 16 };
    System.out.println(squares[i]);
}
```

Figure 12: *Visual representation of unreachable objects created by the loop shown below. The objects not connected to main's activation record will slowly drift to the right.*

In a language such as C/C++ that uses explicit memory management, memory leaks can arise if an object is not freed before the last pointer to this object is overwritten or deallocated. In our visualization, nodes that are unreachable from either the stack or a global object tend to drift off towards the right because of the gravitational directed force we added to the layout force simulation. Objects that are still reachable are being held via a chain of springs from the root or roots keeping them alive. Thus, leaked objects will appear to be floating to the right. Any drifting object in C is a programming error. A user can see and understand how the leak was created by observing the moment in time at which the object was detached from an object that was still reachable. In Java, such drifting objects become subject to garbage collection (which a user can perform manually using a button we provide).

However, understanding the memory impact of one's code is important even in languages such as Java in order to avoid churn. Churn is the repeated (and unnecessary) allocation of objects. For example, consider the Java code shown in Figure 12. This code repeatedly allocates an array of integers when the loop's body is entered. In our

visualization, this fact becomes clear because the objects representing those arrays do not escape the loop — thus, 'squares', which holds the only reference to each array, will always point to the last allocated instance. As soon as 'squares' is reassigned, the previous instance will drift away. (An author of this paper, who is an experienced Java programmer, was surprised to learn of this aspect of Java when he saw it within the visualization HDPV provided; he initially suspected an error within the vizasm monitor his coauthor had written.)



```
import viz.java.util.HashMap;

public class DegHash {
    public static void main(String []av) {
        HashMap<Object, Boolean> map
            = new HashMap<Object, Boolean>();

        for (int i = 0; i < 8; i++)
            map.put(new Object() {
                public int hashCode() { return 0; }
            }, new Boolean (true));
    }
}
```

Figure 13: *Graphical display of a memory anomaly introduced by a bad choice of a hash function. The figure at the top shows an instance of type viz.java.util.HashMap, which is a clone of java.util.HashMap. After inserting 8 elements and applying a horizontal tree layout, the resulting structure appears deep, rather than tall (as it would be if the hash function distributed entries evenly across the slots used to hold the anchors of each list of colliding hash entries). The snapshot shown at the bottom shows a zoomed view of a HashMap.Entry instance that is part of the HashMap shown above.*

An example of a memory anomaly that can have severe impact on the performance of an application is shown in Figure 13. In this example, a user chose a hash function that incurs a high rate of collisions (in this admittedly contrived example, the rate of collisions is 100%). Consequently, instead of evenly distributing hash map entries across the anchors in the anchor table, all hash map entries will accumulate in a singly-linked list anchored at slot 0. As a result, this hash map will provide linear rather than near-constant lookup time. Such "degenerate maps" have been observed as a source of performance degradation in large application frameworks.

6 Related Work

Price et al [Price et al. 1993] includes a review of pioneering work in the area of software visualization (SV) and provides a principled taxonomy for such systems. HDPV shares features with the data view capabilities of such systems as BALSA [Brown and Sedgewick 1984], Zeus [Brown 1991], Incense [Myers 1983], Pascal Genie [Myers et al. 1988; Chandhok and Miller 1989], ANIM [Bentley and Kernighan 1991], UWPI [Henry et al. 1990], and ObjectCenter [Cen 1991]. It also shares features with such tools as JInsight [De Pauw et al. 2002], JavaVis [Oechsle and Schmitt 2002], Cacti [Reiss 1997], Jive [Reiss 2003], Jove [Reiss and Renieris 2005] and DDD [Zeller and Lütkehaus 1996]. Tools originally developed for algorithm animation have also been applied to further visual program understanding and debugging, such as the LENS [Mukherjea and Stasko 1994; Mukherjea and Stasko 1993] or VCC [Yates et al. 1996] systems, or as learning aids such as Samba [Stasko 1997]. The ALVIS [Hundhausen and Brown 2005a] system provides data structure visualizations for programs written in a teaching language. The prefuse toolkit has been used for data structure visualization in [Erkan et al. 2007].

Jeliot 3 [Moreno et al. 2004] and jGRASP [Hendrix et al. 2004; Hendrix et al. 2007] are the two currently available systems that are most related to our work. Unlike these systems, HDPV currently lacks integration into an Integrated Development Environment (IDE). Aside from such integration, the jGRASP system supports additional features that can visualize a program's control structure via Control Structure Diagrams (CSD). Jeliot's strengths lie in its ability to visualize the evaluation of expressions by visualizing how expressions map to the grammatical structure of a Java program. We attempted to run the simple examples in Section 4.2 in jGRASP and Jeliot, but were unable to obtain meaningful data structure visualizations that would have allowed a user to grasp the essence of the data structures being built.

Although jGRASP implements animated object viewers, these viewers can visualize only objects (and their descendants) that are rooted in a variable that is currently in scope. Each viewer visualizes the objects reachable from a single variable; as a result, aliases to objects are not considered. Both jGRASP's object viewers and Jeliot abstract away the actual layout of an object in memory. On the other hand, jGRASP's object viewers have built-in knowledge on how to display specific structures such as trees or lists and can therefore provide abstracted views for them. Jeliot, on the other hand, lays out all objects in a linear fashion, which appears to restrict its usability to very small structures. Neither of these tools support the interactivity we provide and both of these tools are restricted to the Java language (or a subset thereof).

7 Future Work and Conclusion

Although we hope that the effectiveness that was demonstrated in previous studies on data structure visualization [Goldenson 1989; Jain et al. 2006] and, albeit it to a lesser extent, in studies that investigated the use of interactive algorithm animations [Hundhausen et al. 2002; Hundhausen and Brown 2005b], will hold for our system as well, we will need to perform an empirical usability study to confirm or reject this hypothesis. Two usability questions are of particular interest: first, we will need to investigate if the interaction options we provide (layouts, repositioning, collapsing and expanding subtrees) are sufficient to allow the successful navigation of larger heaps (say several thousand objects), or if the aliasing relationships between objects make such navigation impossible. Second, we will need to investigate if our focus on concreteness (e.g., drawing objects to scale, displaying all points-to relationships, etc.) helps or obscures a user's understanding of their program's data structures. If so, we could introduce degree-of-interest (DOI) functions that determine the visibility of edges in the object graph. The infrastructure we have built could also be extended to support program or algorithm animation. A scripting interface could add similar functionality as that provided in such systems as TANGO [Stasko 1990].

This paper presented our design and implementation of an interactive, faithful, in-vivo 2D visualization system for native C/C++ as well as Java programs. The focus of our work lies on providing concrete views of a program's stack, heap, and global variables that accurately reflect the content and points-to relationship of all objects manipulated by a program. We have discussed a number of possible use cases related to common program understanding tasks, including the visualization of the effects of programming er-

rors. HDPV's focus on concrete, faithful visualization for programs in multiple languages, and its focus on providing the user an opportunity to actively interact with their program's data set it apart from existing tools.

References

BENTLEY, J. L., AND KERNIGHAN, B. W. 1991. A system for algorithm animation. *Computing Systems 4*, 1, 5–30.

BROWN, M. H., AND SEDGEWICK, R. 1984. A system for algorithm animation. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 177–186.

BROWN, M. H. 1991. Zeus: a system for algorithm animation and multi-view editing. In *Proceedings of the IEEE Workshop on Visual Languages*, 4–9.

BRUNETON, E., LENGLET, R., AND COUPAYE, T. 2002. ASM: a code manipulation tool to implement adaptable systems. Tech. rep., France Telecom R&D, Grenoble, France, November.

BUCHHEIM, C., JÜNGER, M., AND LEIPERT, S. 2002. Improving walker's algorithm to run in linear time. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, Springer-Verlag, London, UK, 344–353.

CENTERLINE SOFTWARE, INC. 1991. *ObjectCenter Reference*. Cambridge, MA.

CHANDHOK, R. P., AND MILLER, P. L. 1989. The design and implementation of the Pascal GENIE. In *CSC '89: Proceedings of the 17th conference on ACM Annual Computer Science Conference*, ACM, New York, NY, USA, 374–379.

DE PAUW, W., JENSEN, E., MITCHELL, N., SEVITSKY, G., VLISSIDES, J. M., AND YANG, J. 2002. Visualizing the execution of Java programs. In *Revised Lectures on Software Visualization, International Seminar*, Springer-Verlag, London, UK, 151–162.

DEWAR, R. B. K., AND SCHONBERG, E. 2008. Computer science education: Where are the software engineers of tomorrow? *CrossTalk, The Journal of Defense Software Engineering 21*, 1 (January), 28–30.

ERKAN, A. S., VANSLYKE, T. J., AND SCAFFIDI, T. M. 2007. Data structure visualization with LaTeX and prefuse. *SIGCSE Bull. 39*, 3 (September), 301–305.

ERNST, M. D., PERKINS, J. H., GUO, P. J., MCCAMANT, S., PACHECO, C., TSCHANTZ, M. S., AND XIAO, C. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program. 69*, 1-3, 35–45.

GOLDENSON, D. R. 1989. The impact of structured editing on introductory computer science education: the results so far. *SIGCSE Bull. 21*, 3 (September), 26–29.

GOPAL, P. 2006. *Top: An infrastructure for detecting application-specific program errors by binary runtime instrumentation*. Master's thesis, Virginia Tech.

HEER, J., CARD, S. K., AND LANDAY, J. A. 2005. prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceeding of the SIGCHI conference on Human factors in computing systems*, ACM Press, New York, NY, USA, 421–430.

HENDRIX, D. T., JAMES, AND BAROWSKI, L. A. 2004. An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, ACM, New York, NY, USA, 387–391.

HENDRIX, D., CROSS, J. H., JAIN, J., AND BAROWSKI, L. 2007. Providing data structure animations in a lightweight IDE. *Electronic Notes in Theoretical Computer Science 178*, 101–109.

HENRY, R. R., WHALEY, K. M., AND FORSTALL, B. 1990. The University of Washington illustrating compiler. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, ACM, New York, NY, USA, 223–233.

HUNDHAUSEN, C. D., AND BROWN, J. L. 2005. What you see is what you code: a radically dynamic algorithm visualization development model for novice learners. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 163–170.

HUNDHAUSEN, C. D., AND BROWN, J. L. 2005. Personalizing and discussing algorithms within CS1 studio experiences: an observational study. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, ACM, New York, NY, USA, 45–56.

HUNDHAUSEN, C. D., DOUGLAS, S. A., AND STASKO, J. T. 2002. A meta-study of algorithm visualization effectiveness. *J. Vis. Lang. Comput. 13*, 3, 259–290.

JAIN, J., JAMES, HENDRIX, D. T., AND BAROWSKI, L. A. 2006. Experimental evaluation of animated-verifying object viewers for Java. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, ACM, New York, NY, USA, 27–36.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not. 40*, 6 (June), 190–200.

MORENO, A., MYLLER, N., SUTINEN, E., AND BEN-ARI, M. 2004. Visualizing programs with Jeliot 3. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, ACM, New York, NY, USA, 373–376.

MUKHERJEA, S., AND STASKO, J. T. 1993. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, 456–465.

MUKHERJEA, S., AND STASKO, J. T. 1994. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *ACM Trans. Comput.-Hum. Interact. 1*, 3 (September), 215–244.

MYERS, B. A., CHANDHOK, R., AND SAREEN, A. 1988. Automatic data visualization for novice Pascal programmers. In *IEEE Workshop on Visual Languages*, 192–198.

MYERS, B. A. 1983. INCENSE: a system for displaying data structures. *SIGGRAPH Comput. Graph. 17*, 3 (July), 115–125.

OECHSLE, R., AND SCHMITT, T. 2002. JAVAVIS: automatic program visualization with object and sequence diagrams using the java debug interface (JDI). In *Revised Lectures on Software Visualization, International Seminar*, Springer-Verlag, London, UK, 176–190.

PRICE, B. A., BAECKER, R. M., AND SMALL, I. S. 1993. A principled taxonomy of software visualization. *Journal of Visual Languages & Computing 4*, 3 (September), 211–266.

REISS, S. P., AND RENIERIS, M. 2005. Jove: Java as it happens. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, ACM, New York, NY, USA, 115–124.

REISS, S. P. 1997. Cacti: a front end for program visualization. In *INFOVIS '97: Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, IEEE Computer Society, Washington, DC, USA.

REISS, S. P. 2003. Visualizing Java in action. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, ACM, New York, NY, USA, 57–65.

STASKO, J. T. 1990. Tango: A framework and system for algorithm animation. *SIGCHI Bull. 21*, 3 (January), 59–60.

STASKO, J. T. 1997. Using student-built algorithm animations as learning aids. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, ACM, New York, NY, USA, 25–29.

YATES, B. R. A., QUEZADA, G., AND VALMADRE, G. 1996. Visual debugging and automatic animation of C programs. In *Software Visualisation*, P. D. Eades and K. Zhang, Eds., vol. 7. World Scientific, Singapore, 46–58.

ZELLER, A., AND LÜTKEHAUS, D. 1996. DDD - a free graphical front-end for UNIX debuggers. *SIGPLAN Not. 31*, 1 (January), 22–27.