HDPV: Highly Interactive, Faithful, In-Vivo Runtime State Visualization for Software Programs

Jaishankar Sundararaman

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Godmar Back, Chair
Chris North
Eli Tilevich

June 26, 2008
Blacksburg, Virginia

Keywords: Program Visualization, Visual Debugging, Software Comprehension

Copyright © Jaishankar Sundararaman 2008
HDPV: Highly Interactive, Faithful, In-Vivo Runtime State Visualization for Software Programs

Jaishankar Sundararaman

(ABSTRACT)

Program Visualization systems use graphics and animation to represent the behavior of software programs. These systems represent different aspects of the program such as source code, control flow, data structures, runtime state of the program. Representing the actual runtime state of the program finds its use in a variety of applications including program understanding, visual debugging, and pedagogy. However, existing state-of-the-art program visualization systems are limited in: (1) not providing sufficient interactive capabilities to the user; (2) not faithfully representing the runtime state of the program; (3) not allowing users to apply different layout strategies to the visualization; (4) being tied to a specific programming language.

To address these limitations, this thesis presents HDPV, a program state visualization system that visualizes any C, C++, or Java program. HDPV is based on a canonical state model that represents the memory layout of the program as a graph of memory blocks. It decouples the visualization of the program from the actual programming language in which it is written, thereby making the system language independent. HDPV supports a host of interactive features that allow the user to selectively explore different parts of the program’s runtime state. Novel layout strategies support customization through user interaction. We provide a list of use-cases to show that HDPV can be applied to a wide variety of applications including - but not limited to - understanding programs that use basic concepts in computer science, demonstrating algorithm implementations, and debugging software programs.
Acknowledgements

My first words of gratitude go to my advisor Dr. Godmar Back. I had the pleasure of working with Dr. Back in two capacities, first as a teaching assistant and then as a research assistant. As a teaching assistant under him, I have witnessed him as a teacher who renders service of uncompromising quality. It is during these days that I learnt how to work systematically. As a research assistant, I have witnessed how Dr. Back’s insistence on strict discipline and hardwork helped push his students to the best of their research abilities. It is his sharp criticism that provided a sound foundation for my thesis. Numerous are those instances when I had to run to his office (adhering to his unforgettable ‘door algorithm’) to resolve any issue, no matter how complicated, in a jiffy.

Moments to cherish are the times I spent in the company of Dr. Eli Tilevich, especially during the long nights I spent at the computer science lab at CRC. May it be doubts in \LaTeX, ideas for choosing a name for my thesis, or planning a cross country road trip, Eli’s door was always open. Eli is a person whom any student would think as a friend, philosopher, and guide. He was definitely more than just another member of my thesis committee.

One of my favourite courses at Virginia Tech was Information Visualization. The credit belongs to my other committee member, Dr. Chris North. His unique teaching style in this course helped me develop a special interest for the field of visualization. His valuable ideas formed the cornerstones of my thesis work.
Dr. Ali Butt’s support and guidance helped me get my project in his *Operating Systems* course published. I was also fortunate to take the course on *Advanced Networking* under Dr. Wuchun Feng, who is fondly referred to as Wu.

My friends made my days at Virginia Tech enjoyable and memorable. There was no lack of fun and frolic when I was in the company of Ajith and Lokeya. I still remember the scenic drives we took around the Blacksburg area whenever I needed a break from my thesis work. I thoroughly enjoyed the company of Sriram and Tilo at CRC, while my roommates Lakshman and Ashwin made my stay at home comfortable. Words of thanks to Sneha and Anusha for proof reading my thesis and offering valuable suggestions.

For inspiration from remote sources, I could always count on Sheetal and Janani.

Last, but not least, I would like to thank my parents and my brother for their support and unwavering trust in my abilities.

My respects to Almighty!
Contents

List of Figures viii

List of Tables xii

Chapter 1: Introduction 1

1.1 Motivation .................................................. 2
1.2 Contributions ............................................... 7
1.3 Outline ..................................................... 8

Chapter 2: Background 9

2.1 Information Visualization ................................. 9
    2.1.1 Concepts ............................................ 10
    2.1.2 Graph Visualization ................................. 13
    2.1.3 Frameworks ......................................... 17
2.2 Software Visualization ................................. 20
    2.2.1 Algorithm Visualization ............................. 20
    2.2.2 Program Visualization ............................. 22
2.3 Program Analysis Techniques ......................... 25

Chapter 3: Architecture 32

3.1 Canonical State Model ................................. 33
3.2 Intermediate Language Representation ............. 36
3.3 Monitor

3.3.1 Type Information Extraction
3.3.2 Stack Frame Construction
3.3.3 Tracking Allocation andDeallocation Events
3.3.4 Tracking Assignments
3.3.5 Source Location Extraction

3.4 Visualizer

3.4.1 Representation Strategy
3.4.2 Layout Strategy
3.4.3 Navigation Strategy
3.4.4 Interaction Strategy

Chapter 4: Implementation

4.1 Monitor Implementation

4.1.1 The DWARF Debugging Information Format
4.1.2 Type Information Extractor
4.1.3 Runtime State Monitor
4.1.4 Assumptions and Limitations

4.2 Visualizer Implementation

4.2.1 Shadow State Implementation
4.2.2 Custom Renderers
4.2.3 Layout Groups
4.2.4 Node Link Tree Layout

Chapter 5: Evaluation

5.1 Example Use Cases

5.1.1 Understanding Basic Concepts in Computer Science
Chapter 5: Algorithms

5.1.2 Demonstrating Algorithm Implementations ........................................ 103
5.1.3 Visual Debugging ............................................................................. 108

5.2 Performance ......................................................................................... 110

5.2.1 Performance of Monitor ................................................................. 112
5.2.2 Performance of Visualizer ............................................................... 113

5.3 Current Limitations .............................................................................. 114

Chapter 6: Related Work ........................................................................... 115

Chapter 7: Conclusions and Future Work .................................................. 122

7.1 Conclusions ......................................................................................... 122
7.2 Future Work ......................................................................................... 123

Bibliography ............................................................................................... 125

Appendix A: Source Code for Use Cases ..................................................... 133

A.1 recursion.c ......................................................................................... 133
A.2 memory_leak.c .................................................................................. 134
A.3 cycle_detection.c .............................................................................. 135
A.4 traverse.cpp ...................................................................................... 137
A.5 alist.cpp ............................................................................................ 138
A.6 reverse_list.cpp ................................................................................ 139
A.7 linked_tree.cpp ................................................................................ 141

Note: All images are property of the author unless otherwise noted
## List of Figures

1.1 Snapshot of HDPV ......................................................... 7

2.1 Focus + Context .......................................................... 11
2.2 Overview + Detail .......................................................... 12
2.3 Brushing and Linking .................................................... 14
2.4 Filtering ................................................................. 15
2.5 Force Directed Layout .................................................. 16
2.6 Sample visualizations produced with prefuse [HCL05] .................. 19
2.7 Snapshot of Bloom [Rei01] ............................................ 24
2.8 Snapshot of Jeliot ........................................................ 25
2.9 Snapshot of JGRASP ...................................................... 26
2.10 Architecture of Top [Gop06] .......................................... 30

3.1 System Architecture of HDPV ......................................... 33
3.2 Code Sample: Multiple types associated with a memory block ........ 34
3.3 Representation of memory blocks ..................................... 35
3.4 Code Sample: Pointer assignments to offsets within a memory block . 35
3.5 Sample ‘Typedef’ message .............................................. 37
3.6 Sample ‘Memalloc’ message ........................................... 38
3.7 Sample ‘Memfree’ message ............................................. 38
3.8 Sample ‘Putfield’ message ............................................. 39
3.9 Sample ‘Showas’ message .............................................. 39
List of Tables

3.1 Basic types showing different data types .................................. 36
3.2 Interaction controls ................................................................. 58
4.1 Function variable information collected from DWARF information .... 75
4.2 Layout Groups and their function .............................................. 90
5.1 Monitor Performance ............................................................... 113
5.2 Performance of Visualizer ......................................................... 113
6.1 Price et. al’s classification of program visualization systems .......... 116
6.2 Comparison of existing program visualization systems with HDPV ... 121
Chapter 1

Introduction

Program Visualization uses computer graphics and animation to represent source code, flow of control, and data of software programs. Program State Visualization systems visualize the run time state of a program. Runtime state representation finds its application in such areas as visual debugging and program understanding. It is also used widely in teaching introductory programming courses to students. However, program visualization systems currently suffer from a number of limitations: they have limited interactive capability, do not visualize the program in-vivo, and are often tied to a specific programming language.

Our work presents HDPV, a program state visualization system that solves many of the limitations found in current program visualization systems. HDPV visualizes the running state of any program that uses stack and heap for managing the memory layout of the program. Languages such as C, C++, Java, and C# follow this model. Through a set of illustrating use-cases, we prove that HDPV can be applied in educational as well as in software engineering applications.
1.1 Motivation

Our idea of building a program visualization system stems several motivating factors.

Understanding Extent, Scope and Pointer Arithmetic: Our experience with undergraduate students taking CS3204 (Operating Systems) at Virginia Tech has shown that many students lack a basic understanding of core concepts in C/C++, such as extent, scope, and pointer arithmetic.

Extent and Scope: Students do not have a clear understanding of the effect of creating variables in different memory regions such as the stack and the heap. They often erroneously create variables locally within a function to represent data structures that must be preserved across function calls because they do not realize that local variables allocated on the runtime stack go out of scope once the function is exited. Tools that visualize the memory regions in which variables are created can help the students in developing a solid understanding of these concepts.

Pointer Arithmetic: Students use incorrect pointer arithmetic, leading to erroneous access past the end of the valid memory regions. Visual tools that illustrate the effect of pointer arithmetic operations will help the students in understanding this concept correctly.

Identifying Array Overrun Errors: Array overrun errors can corrupt adjacent data structures when programming in languages such as C/C++. If the array and its adjacent data structures are visually represented as objects placed adjacent to each other, array overrun errors can be identified easily.
Detecting Memory Leaks: Memory Leaks can cause problems in long running software systems. If memory is allocated frequently and not released after its use, the system will eventually run out of memory. Since C/C++ does not have automatic garbage collection, the programmer must free the memory that she allocated. If relationships between memory regions are represented visually, leaked memory regions will be shown as objects not linked to any other memory region, signifying a memory leak. Visual tools will be useful to programmers not only in identifying the leaked memory regions but also in detecting the source of the memory leak.

Understanding Recursion: Recursion is a widely used, but difficult to understand, concept in Computer Science [AVI00]. Research has shown that concrete conceptual models perform better than abstract models in teaching the concept of recursion to novice programmers [WDB98]. These concrete conceptual models provide a analogy of the target system in terms of another system and show the internal process of the system at an appropriate level of detail. Stack Simulation [Gre87] is a concrete conceptual model that traces the recursive call with reference to the system stack mechanism used when implementing recursion in a programming language. Visual tools that portray the concept of recursion through stack simulation serve as effective teaching aids to explain the concept of recursion to students.

Identifying Aliasing Effects: Aliasing of objects can lead to many hard to detect bugs in software programs. Aliasing results if one object is reachable through more than one reference. If the object is manipulated by one of the referrers, the state of other referrers may be compromised. Improper use of aliases can result in creation of dangling pointers and double free errors that can lead to unexpected results. In such cases, tools that visually identify all objects that hold a reference to a particular object will be helpful in tracking issues that arise due to aliasing.
Demonstrating Algorithms: Algorithms suitable for demonstrations are traditionally crafted using algorithm animation systems. With these systems, the user creates animations by annotating the source code with suitable calls to the animation system. This activity requires considerable effort on the part of the instructors who create these animations; it has been shown to be one of the obstacles for the widespread adoption of algorithm visualization technology [HDS02]. Systems that automatically generate demonstrations of data structures and algorithms from programs that implement those algorithms may lead to easier adoption of the visualization technology for pedagogical purposes.

Effectiveness of Visualization Technology

Extensive research has been carried out on the effectiveness of visualizations on the learning abilities of individuals. Shaffer et al. found that students find it difficult to transition from abstract learning models such as text books to concrete implementations in computer programs [SHY96]. They advocate the use of visualizations to bridge this ‘concept-gap’. Studies have shown that visualization technology greatly enhances the process of software comprehension and is useful in visual debugging [BDM97]. To take advantage visualization in program understanding, teaching, and debugging, many program visualization tools have been developed. Two tools that closely relate to our work are Jeliot [MMSBA04] and jGRASP [HJB04]. However, they fail to realize their full potential impact because they suffer from a number of limitations.

Limitations of Existing State of Art

First, existing tools are limited in the level of interactivity they provide to the users. Users cannot interact with the visualization to engage themselves in the learning process. Because current display devices have limited amount of screen space, not all parts of the visualization
can be seen in their entirety. This limitation requires the use of information visualization techniques such as zooming, filtering, and panning to explore relevant details and suppress or exclude irrelevant parts of the visualization. But current tools do not provide such capabilities to the users, thus limiting the ability of the tool in providing insights to the users.

Second, existing tools do not provide the capability to the users to apply different layout strategies to subsets of the program’s state. Users cannot move objects to different positions within the visual space or apply custom layout strategies. This restriction limits the capability of the users in visualizing a set of objects according to their interest. For instance, users might want to layout a set of nodes in the form of a tree, but current tools do not provide such a capability to the user.

Third, current tools visualize the state of the program in-vitro rather than in-vivo. In-vivo refers to the activities that happen within the actual system, whereas in-vitro reflects activities carried out in an artificial environment. In such an artificial environment, certain facts of the program might be hidden or otherwise misrepresented to the user. Consider an array overflow bug which corrupts values stored in adjacent memory locations. Unless the visualization tool reflects the actual runtime layout during the execution, it will not be able to depict this bug visually.

Fourth, current tools are tied to a particular programming language particularly Java. One of the important transitions that occurred in the early years of this decade is the paradigm shift from procedural languages such as C towards object oriented programming languages such as Java in introductory programming courses [Mag, Hon98]. This trend had a tremendous positive impact by allowing the students to design programs involving data structures and algorithms without having to worry about pointers and memory management. Yet, shielding the students from the internal details of their programs leads to well-documented difficulties in subsequent systems and architecture courses that use C or C++ and has led
to anecdotal evidence of employer dissatisfaction [DS08]. These difficulties can be reduced if the visualization tool applies the same technique to Java as well as C/C++. Using such a tool, Java’s reference semantics can be compared with C’s pointer semantics, Java’s class can be compared with C’s struct etc. Therefore, we need a uniform tool that applies the same visualization ideas to explain similar concepts in either language.

To overcome these limitations, program visualization tools must be designed with the following goals in mind:

1. **Interactivity**: Support a high degree of interactivity to enable user exploration of the visualization

2. **Fidelity**: Express the run time state of the system in-vivo rather than in-vitro

3. **Scalability**: Design effective information visualization strategies to handle large program structures in a limited amount of display space

4. **Language Independence**: Develop the visualization system language independent, making it applicable to any programming language that follows the stack/heap based runtime model

5. **Persistence**: Save the visual state of the program to a persistent medium and restore the state when desired

With these goals in mind, we have developed HDPV, a tool for interactive 2D graph-based program state visualization. HDPV consists of language dependent monitors that produce information about the state of the program. A language independent visualizer takes the program information produced by the monitor as input and displays the state of the program visually. The visualizer represents the state of the program as a set of stack, heap, and global structures. HDPV’s strength lies in the interactivity it can provide to the user. The user
has the ability to control the sequence of steps of the program, zoom in on areas of interest for closer visual inspection, hide or unhide parts of the visualization or arrange the objects according to her need and interest. The user can apply different layout strategies to position the visual objects displayed. She can assimilate the details of the visualization at her own pace in the form of a animated sequence. The visualizer also provides the flexibility to save the current visual state to a persistent medium and resume later from the point at which it was saved.

Figure 1.1: Snapshot of HDPV

Figure 1.1 gives a snapshot of the HDPV system we developed.
1.2 Contributions

• Design and implementation of a Program Monitor that uses binary instrumentation to extract program state information for C/C++.

• Design of a Canonical State Model to represent the memory layout of the program as a graph of visual objects.

• Design and implementation of a interactive 2D Visualizer that produces a faithful, in-vivo, graph based visualization of programs written in multiple languages.

• Design of novel layout strategies that enable visualization customization through user interaction.

1.3 Outline

Chapter 2 explains the concepts related to our work. Chapter 3 describes the architecture and the design aspects of our system. Chapter 4 describes the implementation details of the system. Chapter 5 describes how we evaluated our system. Chapter 6 compares our system with related work. Chapter 7 describes the future work and conclusion.
Chapter 2

Background

This section introduces concepts used in this work. Section 2.1 introduces the information visualization concepts for designing effective program visualization systems. Section 2.2 discusses software visualization and its two major categories, which are algorithm visualization and program visualization. Section 2.3 discusses how to collect the data for visualization from running programs.

2.1 Information Visualization

Information Visualization is the process of representing abstract data in a visual form that improves the understanding and processing of information. Information visualization concepts form the basis for designing effective program visualization systems.
2.1.1 Concepts

Navigation Strategies

Navigation strategies such as Zoom and Pan, Focus + Context, and Overview + Detail determine how users can navigate to different parts of the visualization.

**Zoom and Pan:** Zooming increases or decreases the magnification of the visual items. Users dynamically zoom into the information space to reach relevant details of interest and consequently zoom out back to their original view. Panning refers to the smooth movement of the screen to bring a set of visual items into the viewing coordinate system. Zoom and pan allows the exploration of parts of the visualization in detail and the exclusion of irrelevant details from view. Zooming strategies fall into two types:

- **Geometric Zooming:** Provides a graphical magnification of the information.
- **Semantic Zooming:** Provides different information content at different zoom levels.

The disadvantage of the zoom and pan approach is that the surrounding context is lost when zoomed deep into the visual space.

**Focus + Context:** The Focus + Context strategy enables viewers to explore a focus area of the visualization in detail but at the same time maintain an impression of the surrounding information, known as *context*. Here, the focus region is expanded within the overview context. Fish-eye menus [Fur86] and Bi-focal displays [ATS82] are applications of Focus + Context. These systems achieve focus as well as context by using distortion functions that distort the background to keep it in context while providing details of the focussed area.
Although this strategy is space efficient and connects details with context, the distortion may cause user disorientation.

Figure 2.1 shows a fish eye menu, an application of Focus + Context strategy.

![Fish Eye Menu](http://www.cs.umd.edu/hcil/fisheymenu/fisheymenu-demo.shtml)

**Figure 2.1:** Focus + Context: A fisheye Menu (Source: http://www.cs.umd.edu/hcil/fisheymenu/fisheymenu-demo.shtml)

**Overview + Detail:** Overview + Detail provides multiple views of the same visual data. The **Overview** provides a broad overview of the entire visualization space whereas the **Detail** view shows a selected area of the visualization space. A field-of-view indicator in the Overview provides visual feedback of the portion of the visualization space that is currently covered by the Detail view. Overview + Detail is commonly used in map and imaging software. See-soft [ESJ92] is an software visualization application that uses the concept of
Overview + Detail. In See-soft, the Overview window provides a visual overview of the entire source code and a Detail view displays lines of code selected through the overview window.

Figure 2.2 shows Google Maps webpage displaying the concept of overview and detail.

![Figure 2.2: Overview + Detail (Source: http://maps.google.com/)](image)

**Interaction Strategies**

Interaction strategies are designed to enhance the scalability of visual information. Because of the limited size of the display screen, not all parts of a visualization can be displayed to the user. Interaction strategies such as selecting, brushing and linking, and filtering help the users to explore relevant parts of the visual space interactively.

**Selecting:** Selection is an interaction strategy in which users can select or pick a set of data items that are of interest to them. This technique is commonly used to highlight a set of objects to differentiate it from the rest of the objects. When the selection strategy is used to reveal more details interactively, it is called *Details on Demand*. *Details on demand* is
used in the design of tooltips.

**Brushing and Linking:** Linking is used to interactively establish a relationship between multiple views of a visualization. The technique is used for visually indicating which parts of the data correspond to one another. Linking is commonly used with the brushing technique and collectively termed as *Brushing and Linking*. If users brush (or select) a region of the data display, related data in other views are highlighted to enable users to establish the relationship between the different views.

Figure 2.3 shows the Spotfire application which employs the *brushing and linking* strategy. The figure shows that when a data point is selected by clicking on the item (brushing), the corresponding details about the item are updated in the information view.

**Filtering:**

Filtering is an interaction technique that minimizes the amount of data that is displayed by employing filters on the data. Filtering is commonly activated through range sliders. *Dynamic Query* is a technique in which filters are applied dynamically and the results are continuously updated to the user.

Figure 2.4 shows Spotfire application using sliders and checkboxes to implement dynamic queries.

### 2.1.2 Graph Visualization

Graph visualization is a technique in which data is represented as a graph structure and visualized. Graph visualization has been used for solving several problems. For instance, *GEVOL* is a tool which visualizes the evolution of software. This system extracts
Figure 2.3: Brushing and Linking: Spotfire window displaying a selection and the corresponding information view reflecting the selection information about a Java program stored within a version control system and displays the extracted information using a temporal graph visualizer. In his paper, *Program state visualization tool for teaching CS1* [Sep04], Seppala visualizes method invocations and references as a graph.

One of the challenges involved in graph visualization is the automatic layout of graphs. The next section discusses two graph layout algorithms that are used for the automatic layout of graphs.
Figure 2.4: Filtering: Use of *Dynamic Query* technique in Spotfire application

**Graph Layout Algorithms**

Graph layout algorithms determine how the nodes and edges of a graph should be positioned on the 2D coordinate system. The following section describes two graph layout strategies namely, *Force Directed Layout* and *Tree Layout*.

**Force Directed Layout**: Force directed layout algorithms view the graph as a *physical system* where nodes of the graph represent the bodies of the system and the edges act as springs between the nodes. The nodes are subjected to forces such as electrical or gravitational forces. These forces move the nodes until they come to an equilibrium state in which the system possesses minimum energy.

Figure 2.5 shows a sample set of objects laid out using a force directed layout algorithm.
A number of layout algorithms are based on the force layout model. The *Fruchterman and Reingold algorithm* [FR91] considers a spring-like force between every pair of neighbors. In this algorithm, two principles determine the layout of the graph:

1. Nodes connected by an edge should attract each other.
2. Nodes should not be drawn too close to each other.

The algorithm uses the concept of *temperature* to control the movement of nodes. *Hot* nodes move faster. The *temperature* controls the step width of node movements and acts as the termination factor for the algorithm.

Other algorithms used for force directed placement include algorithms by *Kamada and Kawai* [KK89] and *Eades* [Ead84].
Tree Layout: Tree Layout algorithms work on tree-structured graphs. A tree layout algorithm considers the parent-child relationship between the nodes of the graph when laying out the nodes of the graph. Many tree layout algorithms have been proposed that draw rooted ordered trees of unbounded degree. Walker proposed a tree layout algorithm [W90] in which the y coordinate of the node corresponds to the level of the tree so that the hierarchical nature of the tree is displayed. Buchheim et. al improved the Walker’s algorithm to run in linear time [BJL02].

Hierarchical layout, radial layout, and circular layout are other graph layout algorithms.

2.1.3 Frameworks

Visualization frameworks provide ready-made modules for building visualization applications. This subsection describes two examples: InfoVis toolkit and prefuse.

The InfoVis Toolkit

The InfoVis toolkit [Fek04] supports the creation, extension, and integration of advanced 2D information visualization components into Java 2D Swing [Micb] applications. It provides a framework for managing data structures such as tables, graphs, and trees. The toolkit provides support for scatter plots, time series, parallel coordinates, tree maps, node link diagrams for trees and graphs and adjacency matrices for graphs.

InfoVis toolkit’s key features include [Fek04]:

- Generic data structures suited to visualization
- Specific algorithms to visualize these data structures
- Mechanisms and components to perform direct manipulation of the visualization
- Mechanisms and components to select, filter and perform generic information visualization tasks

- Components to perform labeling and spatial deformation

The toolkit is designed using five main abstractions: tables, columns, visualizations, components, and input/output modules. **Tables** provide the underlying data structures for the framework. Tables consist of named **columns** plus meta data. **Visualizations** transform the semantic attributes stored in the data structures into visual structures. The **Components** supported are dynamic queries, filters, selection, sorting, and visual attributes manipulation. **Input/Output** modules render the visual objects into a graphical context.

Developing an application using the Infovis toolkit involves using the classes provided by the toolkit. Infovis allows development of small but powerful applications. For instance, implementing a *Parallel Coordinate* visualization for multi-dimensional data set requires only around 96 lines of code using the toolkit.

**Prefuse Visualization Framework**

Prefuse [HCL05] is a visualization framework for creating rich, interactive visualizations for both structured as well as unstructured data. Prefuse is written entirely in Java and is based on the Java 2D graphics library [Mica]. The strength of prefuse lies in its ability to create rich visualizations by assembling the building blocks provided by the framework. Prefuse provides support for data structures such as graphs, trees, and tables and supports multiple layout algorithms, interaction features, animation capabilities, and integrated search features.

Figure 2.6 shows a set of sample visualizations produced with prefuse.

Prefuse’s design is based on the *Information Visualization Reference Model* [Chi99]. This model decomposes visualization design into stages, which include the representation of ab-
Abstract data, mapping of the data into an intermediate visual form (filtering), processing the visual abstraction, and finally mapping the visual abstraction into interactive displays (rendering).

During the first stage, the abstract data is represented in a canonical form using the data structures provided by prefuse.

During the filtering stage, the abstract data is converted into a representation suitable for visualization. Visual attributes are created for the data items selected for visualization. These visual items include properties such as location, color, and size.

Processing of visual items is performed through modules known as Actions. Actions update the visual items. Action modules implement one of the three types of actions. Filtering actions control the entities that should be considered for further processing. Assignment actions set visual attributes such as location, font, color, and size. Animator actions interpolate...
visual attributes between starting and ending values to implement animation capabilities. The visual items are drawn to the screen using Renderers. Renderers read the visual attributes of an item to control the way in which the item is displayed on the screen.

Writing applications with prefuse involves representing the data for visualization using appropriate data structures, setting up the renderers, creating actions to process the visual items and finally setting up a display object to present the visualization to the screen.

HDPV is built on top of the prefuse framework.

2.2 Software Visualization

Software Visualization uses animation and graphics to represent computer programs, algorithms, source code, and control flow. The term software visualization can be broadly classified into two categories: Algorithm Visualization and Program Visualization.

2.2.1 Algorithm Visualization

Algorithm Visualization is the process of visualizing algorithms and their associated data structures. The animations are created using algorithm visualization systems that allow the creation of either static or dynamic visualizations from ready-made visual objects. The visual descriptions are at a higher level of abstraction that concentrate on the description of algorithms rather than the lower level details of the program.

Algorithm animation systems are used mainly for teaching and evaluating algorithms. One of the earliest algorithm animation systems was BALSA [BS84]. Since then, many algorithm visualization systems such as Tango [Sta90], Zeus [Bro91], and Polka [SK93] have been built. Balsa [BS84] annotates an algorithm with markers that specify which parts of the algorithm
to visualize. Balsa calls these parts of the program *interesting events*. The Balsa system contains multiple views, which are notified when events of interest occur in the algorithm. Users can zoom in and out, and can pan the animation views. Users can control the speed at which the algorithm is run and also set break points in the code.

**Tango** [Sta90] maps program data to an animated representation using a finite state automaton. In Tango, a program event can be mapped to different display events depending on the state of the automaton. Tango has since been replaced by Polka [SK93].

**Zeus** [Bro91] is a conceptual framework for building graphical user interfaces. It associates multiple client-defined views with a set of events of interest generated by the algorithm. The views of the system portray the events in the form of animated pictures as the algorithm is executed. The system also provides the ability to the user to specify parameters that control the animation when events of interest are encountered.

**Effectiveness of Algorithm Visualization Systems**

Algorithm visualization systems are a compelling alternative to other forms of instruction such as static images in textbooks or oral presentations through lectures. However, it has not been conclusively shown that algorithm visualization systems are effective in practice. Gurka and Citrin in their paper on *Testing Effectiveness of Algorithm Visualization* [GC96] state that there is not much evidence for the effectiveness of algorithm animation tools which they attribute to experimental flaws. Hundhausen and Douglas found no significant difference in learning outcomes between students who create visualizations on their own from students who were active users of predefined visualizations [HD00]. The authors also point out that using detailed *high fidelity* visualizations, which illustrate an algorithm for general input and is drawn with the quality that matches that of textbook figures, is cumbersome. The authors recommend using sketched *low fidelity art forms* instead, which work for limited
Also some evidence suggests that algorithm visualizations may not be practically effective, there is also a growing body of evidence that suggests that visualizations have a positive impact on student learning. Lawrence found that students who interacted with the animation in a laboratory session performed better than students who viewed prepared examples or no examples [LBS94]. Narayanan and his students designed a Hypermedia Visualization System, Halviz [HNH02] in which animations are embedded in a knowledge based hypermedia environment and showed the superiority of the approach compared to traditional methods of teaching. Other experiments suggest that algorithm visualization requires increased interaction with the animation to reap the benefits of the visualization technology. Hundhausen in his paper, A Metastudy of Algorithm Visualization Effectiveness [HDS02] proves that cognitive engagement with the animation greatly improves learning outcomes.

### 2.2.2 Program Visualization

Program Visualization is the process of visually representing a program in execution. Whereas algorithm visualization systems depict the working of algorithms, program visualizations systems depict the working of programs in general and are not limited to any particular algorithm. These systems portray information at a lower level of detail compared to algorithm animation systems. Source code, algorithms, data structures, dynamic call chains, object relationships are some of the activities depicted by program visualization systems. Program visualization systems are used for teaching, understanding program behavior, visual debugging, and evaluating program performance.

Designing a program visualization system involves the following challenges:

1. How should data be collected for visualization?
2. What aspect of the program should be considered for visualization?

3. What visual representation should be selected for depicting the program behavior?

4. How should program data be depicted in a limited screen size?

The effectiveness of program visualization systems is highly dependent on the approach used for handling each of these challenges. Existing program visualization systems include Bloom [Rei01], Jeliot [MMSBA04] and jGRASP [HJB04].

**Bloom** is a visualization system developed at Brown University that defines a comprehensive framework aimed at general software understanding. Bloom can produce 2D and 3D visualizations for programs written in C, C++ or Java. It collects information from the program in the form of program traces and performs data analysis to identify which details of the program to visualize. It presents the information collected to a information visualization framework that produces multiple high-density views of the information collected. Bloom’s visual query language lets users specify what should be visualized through a graphical front end. Figure 2.7 shows sample visualizations produced with Bloom.

**Jeliot** is a tool for the visualization of Java programs. Jeliot visualizes method calls, assignments, and object allocations. Animation of programs in Jeliot takes place in a visual space known as a *Theater* that is divided into 4 areas: the *method area* where the methods are shown, the *constant area* where constants and static variables are displayed, the *object area*, where allocated objects and arrays are displayed, and the *expression evaluation area* where expressions are evaluated.

Jeliot is targeted at 2 groups:

1. Educators who wish to teach students using suitable animations of Java programs.

2. Students who wish to construct their own programs and visualize the execution of those programs.
Figure 2.7: Snapshot of Bloom: Sample visualizations produced with Bloom [Rei01]

Figure 2.8 shows how Jeliot visualizes a Fibonacci program written in Java.

**jGRASP** is a lightweight development environment that can automatically generate visualizations of software programs. It is designed with the purpose of representing the underlying algorithms and their associated data structures. It produces Control Structure Diagrams (CSD) [IBHT96] for languages such as Java, C, C++, Objective C, Ada, and VHDL, Complexity Profile Graphs (CPG) [YHCU05] for Java and Ada, and UML diagrams for Java. jGRASP shares many properties from both algorithm as well as program visualization systems.

jGRASP is designed to represent the program at multiple levels of detail. The source code view provides an animated view of the line of code that is being executed. The object viewer displays the values held by an object’s member variables while another viewer displays a
Program visualization systems require program information collected from the running program. There are several ways in which data can be collected and used for visualization.

In one method, the data collection step overlaps with the process of visualization. In this method, program monitoring is tightly integrated with the actual visualization system itself. Systems such as jGRASP fall under this category. In jGRASP, the program is placed in
Figure 2.9: Snapshot of jGRASP: jGRASP visualizing a binary tree program

a workbench and executed. As events of interest are reported, the visualization state is reflected accordingly.

Alternatively, data collection may be separated from the process of visualization. In this method, the data collected by analyzing the program is converted into an intermediate language representation which serves as input to the visualizer. Systems such as Jeliot [MMSBA04] fall under this category. In Jeliot, the program information is converted into a representation known as MCode [Gar05]. Because MCode contains semantic information about the program, the visualizer can produce different visualizations of the same underlying program information.

Program information can be obtained using a variety of techniques. The exact choice of technique depends on the requirements imposed by the system under design. The following
specific requirements influenced the choice of program information extraction technique for HDPV:

- **In-Vivo Representation**: HDPV aims to provide a faithful in-vivo representation of the program. Therefore, the information extraction technique must preserve the runtime state that is seen by the program as it executes normally.

- **Efficiency**: The system must monitor the complete state of the program efficiently.

- **Exhaustive Information**: Users must not be burdened with providing inputs about the information to be extracted. Thus, exhaustive information must be collected by the monitoring tool.

- **Accessibility to Source Code**: The system must have the capability to operate on any native binary without depending on access to the source code.

The following sections discuss how existing program analysis techniques fit our set of requirements.

**Debugger Interfaces**

Debugger interfaces provide the infrastructure to build end-user debugging applications. An example of a debugger interface is the interface provided by the Java Platform Debugging Architecture (JPDA). JDPA is a multi-tiered debugging architecture that allows tool writers to create debugger applications. As part of JDPA, the Java Virtual Machine (JVM) exposes a debugging interface known as Java Virtual Machine Debugging Interface (JVMDI). This interface provides a way to request information about the program, create actions such as setting a break point and provides notifications of events. JPDA is unsuitable for our purposes for 2 reasons. JPDA does not expose the exact stack layout to the external environment. Moreover, no notifications are generated when assignments to arrays and local
variables occur. Therefore, we would have to track the state of the program manually by comparing the current value of a variable with its previous values to decide if an assignment has occurred.

Systems such as jGRASP and Javavis [OS01] uses this interface to access program state information. The Data Structure Visualizer [Cos03] program visualization system uses Visual Studio’s debugging interface to collect program information.

**Use of Language Constructs**

Program information can be collected using a variety of techniques. These techniques include aspect oriented techniques that use point-cuts to identify instructions in a program’s execution sequence, compiler modifications that emit additional instructions at compile time that collect program information, use of preprocessor macros, or taking advantage of the *overloading* feature provided by some programming languages. For instance, the C++ `new` operator can be overloaded such that creation of new objects in the program can be tracked. The disadvantage of such an approach is that it might not be feasible to track all the activities in the program.

**Static Instrumentation**

Static Instrumentation involves instrumenting the program before it is loaded into the memory for execution. The instrumentation can be done either at the source level or at the binary level.

Source to source transformation systems read the source code of a program and convert the program into an intermediate representation such as an abstract syntax tree (AST). Using this representation, the program is converted into another program. For example, CIL [NMRW02] is a source to source transformation library for the C language. However,
source to source transformation systems require access to source code. Moreover, since they modify the program at the source level, the runtime state of the program might be different from what the program would experience had it not been subjected to the transformation.

Instrumentation can also be performed directly on the binary. In this method, the native binary is parsed and rewritten into a new binary that has instrumentation code inserted into it. This modified binary is then executed. Systems such as ATOM [SE94] and EEL [LS95] follow this approach. These systems suffer from the disadvantage that since there is no way to know in advance how the program will behave at runtime, it is difficult to instrument selectively. Moreover, the runtime state of the instrumented program may differ from the original program.

Dynamic Instrumentation

Dynamic Instrumentation instruments the program’s binary at runtime. Dyninst [BH00], Pin [LCM+05], Valgrind [NS07], Top [Gop06] etc. are examples of dynamic binary instrumentation tools. Among them, Dyninst, Valgrind, and Top are instrumentation frameworks that are designed as a plugin architecture. These systems use several approaches to instrumenting binaries. Dyninst modifies the original code in memory by inserting ‘trampolines’. Trampoline based systems can be highly inefficient because trampolines have to be inserted even for a simple single instruction modification particularly on variable length instruction set architectures.

On the other hand, systems such as Pin [LCM+05] use translation to perform instrumentation. Pin adds additional code to make the original state of the registers known to the instrumentation code. Therefore, this approach preserves the original runtime state.
**Top Dynamic Instrumentation Framework:**  *Top* is a dynamic binary instrumentation framework that operates on top of *Pin* \[LCM^*05\]. *Top* defines a set of high level APIs that simplifies the instrumentation of C/C++ programs. *Top* is based on an event driven architecture that allows clients to register for events of interest. Once the registered events occur, the clients are notified through callbacks.

Figure [2.10] shows the architecture of the *Top* framework.

![Figure 2.10: Architecture of Top](Gop06)

The following events are currently supported by the *Top* framework.

**Function Call/Return Event:** Clients can use Function Call / Return events to be notified by the framework when the program enters or exits a particular function. The framework also provides information about the parameters that were passed to the function.
when a Function Call event occurs and about the return value of the function call when a Function Return event occurs.

**Memory Access Event:** A Memory Access Event is triggered when a memory region that is registered with the framework is read from or written into. The framework provides information about the type of event, which can be a memory read or a memory write, along with the size and address of the memory access.

In addition to these events, Top also provides a LostReference event and Dereference event. A LostReference event is generated if there are no references to a memory block registered for tracing. A Dereference event is generated when a pointer is dereferenced for read or write access.
Chapter 3

Architecture

HDPV consists of two modules, Program Monitor and the Visualizer. The system is based on a canonical state model that represents memory regions as a graph in which nodes represent the memory regions and edges represent the relationship between the memory regions. Section 3.1 explains the canonical state model.

Figure 3.1 shows the architecture of HDPV.

The Monitor is responsible for collecting information about the runtime state of the program. The Monitor is specific to a particular language. The Monitor consists of two modules, Type Information Extractor and Run time State Monitor. The Type Information Extractor extracts information about the types used in the program. The Runtime State Monitor monitors the program for changes to the runtime state and reports the run time state in the form of program events to the visualizer. The design of the Program Monitor is explained in Section 3.3.

The Visualizer is responsible for visualizing the state of the program. It consists of a graphical front end that renders the run time state information as a graph of nodes and edges. The visualizer uses the type information produced by the Monitor for representing
the memory regions created by the program. The program events generated by the Monitor cause changes to the visual state that is represented by the Visualizer. The Visualizer is not tied to any particular programming language. The design of Visualizer is explained in Section 3.4.

The Program Monitor communicates with the language-independent Visualizer through an intermediate, XML based language which is explained in 3.2.

### 3.1 Canonical State Model

HDPV visualizes the state of the program as a graph of nodes and edges. In order to view one-dimensional, byte-addressable memory as a graph of nodes and edges suitable for a two dimensional layout, we designed a Canonical State Model. Nodes represent the different memory regions of the program such as stack, heap, and global. Edges represent
the relationship between these memory regions created through pointers in languages such as C/C++ and through references in languages such as Java.

*The Canonical State Model* splits the memory regions of the program into intervals known as *Memory Blocks* that form the nodes of the graph that is being visualized. Nodes in the *Canonical State Model* are typed. Multiple type definitions can be associated with a particular node to support languages that allow pointers to an object to be cast to different types.

To understand the need for multiple type definitions, consider the code shown in Figure 3.2 that creates a piece of memory referenced by a character pointer. Initially, the memory pointed to by this pointer is interpreted as an array of characters. Subsequently, this memory is assigned to a pointer of type *record*. Afterwards, the memory pointed to by the pointer to *record* must be interpreted as a object of type *record*. Figure 3.3 illustrates this idea.

```c
struct record {
    int a;
    float b;
};
void main() {
    char *str = (char *)malloc(8);
    struct record *ptr = (struct record *)str;
}
```

Figure 3.2: Code Sample: Multiple types associated with a memory block

The edges of the graph that connect the nodes represent the relationship between the different memory blocks. Edges are directed from the source to the target memory block. Languages such as C/C++ provide a way to create pointers that may point to arbitrary offsets within a memory block. This situation is illustrated in the code snippet shown in Figure 3.4. In this example, pointer arithmetic is performed on a character pointer as a result of which the pointer points to an offset within the memory block:
Figure 3.3: Representation of memory blocks: Memory block represented as two different types, as an array of characters, and as an object

```
void main()
{
    char *str = "Sample";
    str = str + 2; //str points to character m
}
```

Figure 3.4: Code Sample: Pointer assignments to offsets within a memory block

To represent the fact that pointer variables can originate from any offset within a memory block and point to any offset within another memory block, each edge is labeled with two offsets, source offset and destination offset.

To express the types in a language independent manner, we developed a simple type system similar to C99’s `<stdint>` header. Primitive types include signed and unsigned single and multiple byte in integer types and unicode characters. Type constructors can be invoked to create derived types such as arrays, pointers, and record types. Record types consists of a set of fields identified by either a simple type or a derived type.

Table 3.1 shows the list of basic types that are used for representing the data types and their
Table 3.1: Basic types showing different data types

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Representation</th>
<th>Bits</th>
<th>Min Value</th>
<th>Max Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>signed char</td>
<td>8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>uint8_t</td>
<td>unsigned char</td>
<td>8</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>int16_t</td>
<td>short</td>
<td>16</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>uint16_t</td>
<td>unsigned short</td>
<td>16</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int32_t</td>
<td>long</td>
<td>32</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>uint32_t</td>
<td>unsigned long</td>
<td>32</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>int64_t</td>
<td>long long</td>
<td>64</td>
<td>-9,223,372,036,854,775,808</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>uint64_t</td>
<td>unsigned long long</td>
<td>64</td>
<td>0</td>
<td>18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

corresponding range of values.

Finally, the *Canonical State Model* treats activation records of functions as new type definitions so that the visualizer can treat both heap objects and activation records uniformly. The set of formal parameters and local variables are treated as fields of the created activation record type.

### 3.2 Intermediate Language Representation

The Monitor and the Visualizer communicate through a custom designed intermediate language which we describe in this section.

The intermediate language decouples the process of program information collection from the actual process of visualization. The Program Monitor independently encodes the program state information into a series of messages that conform to an intermediate language representation. These messages serve as input to the Visualizer which reads the messages to visualize the program state. The use of an intermediate representation makes the system flexible, allowing any custom program monitor that respects the intermediate language specification to operate with the visualizer.
The intermediate language is built around six messages that are unidirectional from Monitor to Visualizer.

**Typedef**

The *Typedef* message announces a new type to the visualizer. This message consists of the name of the type, its size, and a set of fields. The fields in turn contain a name, size, and their corresponding data type. The *Typedef* message announces both data types as well as activation record types. For activation record types, the field parameter also contains a *kind* parameter which specifies whether the field represents a formal parameter, local variable, or a saved register.

A sample type is shown below. The corresponding XML message generated when processing the type is shown in Figure 3.5.

```c
struct temp {
    int a;
    float b;
    struct temp *next;
}
```

```
<typedef name="temp" size="12">
    <field name="a" size="4" offset="0">
        <int32/>
    </field>
    <field name="b" size="4" offset="4">
        <float/>
    </field>
    <field name="next" size="4" offset="8">
        <pointerto>
            <struct name="temp"/>
        </pointerto>
    </field>
</typedef>
```

Figure 3.5: Sample ‘Typedef’ message
Memalloc

The *Memalloc* message announces any new block of memory allocated in the program. The message contains the base address of the allocated memory, the size of the allocated memory, the type associated with the block of memory and kind that specifies whether the memory allocated belongs to the heap, stack, or global region. Figure 3.6 shows a *Memalloc* message.

```
<memalloc type="stack" address="bffe14dc" size="36">
  <struct name="main"/>
</memalloc>
```

Figure 3.6: Sample ‘Memalloc’ message

Memfree

The *Memfree* message announces any block of memory that is deallocated by the program. The message contains the base address of the memory that is deallocated along with the size of the memory being deallocated. Figure 3.7 shows a *Memfree* message.

```
<memfree address="bffe14c4" size="16"/>
```

Figure 3.7: Sample ‘Memfree’ message

Putfield

Assignments are announced through *Putfield* messages that contain the address along with the value of the variable being assigned. The ‘offset’ field specifies the offset of the memory access within the base address. Figure 3.8 shows a *Putfield* message.
Figure 3.8: Sample ‘Putfield’ message

ShowAs

The Showas message is sent as a hint by the monitor to the visualizer to signal that a particular memory block must be displayed using a specific type. If multiple types are associated with a memory block, this message signals the visualizer to change the type associated with the memory block to the type specified in the message. Figure 3.9 shows a Showas message.

Figure 3.9: Sample ‘Showas’ message

Location

The Location message announces to the visualizer the line in the source code currently under execution. The message carries the line number along with the file that contains the source line. Figure 3.10 shows a Location message.

Figure 3.10: Sample ‘Location’ message
3.3 Monitor

The Monitor collects information about the program to be visualized. It is language dependent. Every program to be visualized needs a language specific monitor that encodes language specific information into language independent format to be provided as input to the Visualizer. This section discusses the different activities performed by the Monitor during the process of program information collection.

3.3.1 Type Information Extraction

The Visualizer needs information about the definition of all types used in the program so that it can interpret the memory regions created in the program as a particular type. Types in our model can either represent data types or activation records of functions. Data type can either refer to simple types such as int, char, float etc or any new types created through programming constructs such as `struct` in C/C++ or `class` in C++/Java.

The Monitor has the responsibility of announcing the definition of all the types created in the program. When the program is loaded for execution, the Monitor extracts all type definitions from the debugging information present in the executable, calculates the size of all the fields within the type and assigns suitable offsets at which the fields are located. It then announces the type to the Visualizer by constructing a `Typedef` message.

3.3.2 Stack Frame Construction

Stack frames are allocated by the runtime system when a function is entered. The Monitor must communicate the stack frame layout to the Visualizer through a `Typedef` message. To achieve an in-vivo representation of the program’s runtime state, the stack frame type
definition should reflect exactly how the stack frame is created in the actual underlying machine architecture.

In a stack based architecture, the following events occur when a function is called:

1. Arguments are pushed onto the stack from right to left.
2. The value of the return address is pushed onto the stack by the caller.
3. The actual function call is performed.
4. The current values of callee saved registers are pushed onto the stack.
5. The stack pointer is adjusted to allocate space for local variables and any temporary memory regions used within the function.

Figure 3.11 shows a typical stack frame layout in a stack based architecture.

The Monitor constructs a stack frame by assembling a type definition with the following elements at the exact offsets at which they reside within the function stack:

1. Formal Parameters.
2. Saved Registers (includes return address and callee saved register).
3. Local Variables.
4. Extra locations allocated in the stack by the compiler (if any).

Construction of ‘invisible’ arguments: In languages such as C++ and Java, ‘this’ holds a reference to the current object. ‘this’ is generally passed as a invisible argument to all the constructors, destructors, and non-static member functions of objects. When constructing the stack frame type definition for constructors, destructors, and member functions, the
Monitor includes the ‘this’ parameter along with other parameters of the function. Thus, ‘this’ appears as another parameter when an activation record is rendered by the Visualizer.

### 3.3.3 Tracking Allocation and Deallocation Events

Programs create and free memory blocks, which causes changes to the program state. The Monitor must announce the creation and destruction of memory blocks to the Visualizer.

**Memory allocation**

Programs create new memory blocks in two scenarios:

1. Entry of a function adds a new memory block to the stack.
2. Calls to memory allocation functions such as `malloc` create memory block on the heap.

The Monitor must announce the allocation of a new activation record when a function is entered. This step involves announcing the size and starting address of the memory block allocated for the activation record. The announced memory block must also be tracked by the Monitor for memory accesses which reflect assignments to local variables. Similarly, memory allocated on the heap using dynamic memory allocators is announced to the Visualizer and subsequently tracked for memory accesses.

New memory blocks are announced to the visualizer through `Memalloc` messages.

**Memory Deallocation**

Programs free memory blocks in two scenarios:

1. Exit of a function frees the memory block representing the activation record of the function. Memory is also freed when the stack frame is unwound during exception handling.

2. Memory deallocation functions such as `free` release memory blocks allocated on the heap.

On function exit, the memory block corresponding to the current activation record of the function is released. Programs use memory deallocation functions such as `free` in C or operators such as `delete` in C++ to deallocate heap memory. The Monitor tracks all calls to such memory deallocators. When a region of memory is deallocated, the Monitor extracts the starting address and size of the memory block being freed and communicates this information to the Visualizer. Systems that perform automatic garbage collection may not have explicit memory deallocation events in the program.
Memory deallocation events are announced to the Visualizer through *Memfree* messages.

### 3.3.4 Tracking Assignments

Assignments cause changes to the program state. The Monitor tracks all assignments that occur in the program. As assignments occur, it communicates the value being assigned to the Visualizer through a *Putfield* message.

Assignments are generally triggered by memory write instructions that move a value from either a source memory location or a register to a destination memory location. Memory write instructions can be `mov`, `push`, or floating point store assembly instructions in compiled C/C++ programs and *putfield, astore, fstore, or istore* instructions in Java bytecode. These instructions may transfer either 8-bits, 16-bits, 32-bits, or 64 bits of data. As memory writes occur, the Monitor extracts the value of the assignment based on the size of the memory write. It then communicates the address of the memory access along with the value assigned to the Visualizer.

### 3.3.5 Source Location Extraction

The Monitor must collect line number information from the executable. This information is required by the Visualizer to provide source code highlighting. Line number information is extracted for the following program events: entry of a function, exit of a function, and assignments. The Monitor extracts the file name and the line number that is currently under execution from the debugging information present in the executable. Debugging information is built into the executable only when compiling the program with `-g` option. Therefore, source location extraction requires that the program is compiled with this debugging option.
3.4 Visualizer

The Visualizer converts the canonical state of the program into a visual graph representation. The Visualizer follows a Representation Strategy to display the nodes and edges of the graph. It follows a Layout Strategy that decides how to layout the nodes and edges of the graph in the visual screen. A Navigational Strategy aids users to navigate the visual space. Several Interaction Strategies help users interact with the visualization.

3.4.1 Representation Strategy

‘Drawn to Scale’ Approach

The nodes that represent the memory regions of the program are drawn to scale. In this approach, the size of the shape representing the nodes is proportional to the size of the memory blocks they represent. We follow the Drawn to Scale approach because visual inspection of the nodes provides valuable information on their relative sizes. A user can identify if one memory region is larger than the other by visually comparing the size of the displayed nodes. Moreover, users can visually identify the space complexity of the memory blocks they create through this approach.

Figure 3.12 illustrates this idea. Here, the temp object contains 3 fields a, b, and c which are of type integer, double, and character. The entire shape of the object and the fields it encompasses are drawn to scale. Here, the width of a and b are 4 and 8 times the size of c, directly proportional to its corresponding data type size.
Representing Memory Blocks

The nodes of the visualizer represent memory regions of the program. A memory region is associated with a type that consists of a set of fields. Fields consist of field names that identify the field and the value that is held by the field. The memory block is drawn such that both the field names and its corresponding field values are displayed to the user.

**Representation of Stack Nodes:** The stack nodes are displayed in a vertical orientation similar to how a stack is conventionally displayed. In vertical orientation, the memory block is split into a set of field blocks which are placed one below the other along the vertical axis. Dotted lines demarcate the different fields. The type name is displayed vertically alongside the node. The field name and values held by the fields are aligned to the center of the field block. The field names and type names are abbreviated if they exceed the size of the rendered shape. Also, the stack nodes are drawn with a particular color code. Figure 3.13 shows how a stack node is represented.

**Representation of Heap Nodes:** The heap regions as well as regions representing global variables are displayed in horizontal orientation. In horizontal orientation, the memory blocks are drawn such that the fields are placed horizontally from left to right. The entire memory
Figure 3.13: Representation of a stack node

block is split horizontally into two regions with dotted lines separating the regions. The top region displays the type name aligned to the center of the block. The bottom region displays the fields’ type and values. If a particular type name or field name cannot be displayed in its entirety, the field is abbreviated and displayed. Similar to stack nodes, heap and global nodes carry a distinct color code to distinguish their type from other memory regions. Figure 3.12 shows a heap node.

**Representation of Arrays:** Arrays represent a collection of objects of the same type. They are represented by dividing the entire memory block into a set of array elements. The array indices form the field names of the array node. Figure 3.14 shows a snapshot of a character array.
Value Representation: Fields contained in memory blocks may be initialized or uninitialized. Uninitialized field values are shown with a ‘?’ in the field representing the field value. This approach allows users to distinguish programming constructs that automatically initialize memory regions from those constructs that do not. The ‘?’ is replaced with the corresponding value once the value is initialized. Once initialized, the values are represented according to their corresponding data type. Pointer / reference values are distinguished from other data types by preceding the value with ‘0x’, for instance, 0xbfe5a3d8. Negative values are shown by preceding the field value with ‘-’ sign.

Representation of Source Code

HDPV supports the highlighting of source code through a source code window. Source code highlighting relates program events with their corresponding source code location, which is helpful in tracking bugs.

Figure 3.15 displays the source code window.
3.4.2 Layout Strategy

Default Layout Strategy

The design of the layout of nodes is based on memory regions they represent. We designed a layout strategy that provides reasonable behavior for small heaps in the absence of any user interaction and that performs well when the user interactively assigns positions to some memory blocks. Nodes may either be fixed or floating. A fixed node’s position will not change when a layout is recomputed. Its coordinates are calculated from the corresponding address of the memory block. Figure 3.16 shows our default layout strategy.

The stack nodes are laid out one below the other similar to how a conventional stack is laid out in memory. The stack nodes are initially fixed although the user can move the node to a desired position.

Similar to stack nodes, global nodes are also fixed. They are laid out such that nodes
representing adjacent memory regions are placed next to each other along the horizontal direction.

The heap nodes are placed initially in the center of the layout but they are not fixed at their position. Once they are placed, they are subjected to a force simulation consisting of multiple forces. A *n-body force* acts on all floating nodes. n-body forces can be positive or negative. We apply negative n-body force to untangle them. Nodes connected by edges are further subjected to spring forces which pull the nodes towards each other. A *drag force* acts in the direction opposite to that of the direction in which a node moves to keep its movement smooth. In addition to these default forces provided by prefuse, two additional forces are added. The first is a *circular wall force* that acts radially outward from the point where the node was initially placed. The second is a directional *gravitational force* that acts horizontally to the right, so that all attached nodes drift towards the right. The size of the
forces governing this layout can be controlled through a preferences panel.

The circular wall force determines the movement of the nodes once they are initially placed in the center of the layout. Initially, all heap nodes are placed in the same position. However, due to the application of the circular wall force that acts radially outward from the center of the layout, the nodes travel radially outward. This force untangles those nodes, preventing the display region from getting congested with a large number of nodes that are placed initially at same position.

The spring forces keep the nodes linked to each other with a spring-like force. This force pulls objects to their referring nodes. When an object is dragged to a different position, the spring force elastically pulls all related objects along with the object.

**Examples of Layout Produced by the Default Layout Policy:** The addition of above forces is based on the design rationale that the default layout produced by our layout strategy should be meaningful even without the user interacting with the visualization. Consider a code snippet shown in Figure 3.17 that creates a list of nodes that form a tree held by a pointer situated on the stack.

Figure 3.18 shows the layout produced by our default layout strategy. This layout was produced without any user interaction.

The code sample shown in Figure 3.19 illustrates a toy linked list implementation and Figure 3.20 shows the corresponding default layout produced without any user interaction.

**Design of Collapse Stack Layout**

The default layout strategy lays out the stack nodes one below the other. If the number of stack frames increases beyond a point, some stack frames might not fit onto the display screen which requires panning to bring those stack frames into the view. To handle this
class Node {
    Node *left, *right;
public:
    Node(Node *left, Node *right) {
        this->left = left;
        this->right = right;
    }
    Node() { this->left = this->right = NULL; }
};

void main() {
    Node *root =
        new Node(
            new Node(
                new Node(),
                NULL),
            new Node(
                new Node(
                    new Node(),
                    new Node()),
                new Node(
                    new Node(),
                    new Node())));
}

Figure 3.17: Code Sample: Simple tree program

Figure 3.18: Default layout produced for a simple tree program
class Cell {
    Cell *next;
    public:
    Cell() { next = NULL; }
    Cell(Cell *next) { this->next = next; }
};
void main() {
    Cell *list =
        new Cell(
            new Cell(
                new Cell(
                    new Cell(
                        new Cell()))));
}

Figure 3.19: Code Sample: Simple list program

Figure 3.20: Default layout produced for a simple list program

issue, we designed the *Collapse Stack Layout* strategy for stack frames. In this strategy, the stack nodes are not laid out one below the other, instead, the successive nodes are placed at a small horizontal and vertical displacement from each other, arranged in the form of a tile. The user can hover over the stack node to inspect the contents of the stack frame. Figure 3.21 shows a snapshot of the collapse stack layout applied to a set of stack nodes.

**Layout Customization:** A control panel provides controls for customizing the layout. The controls can be used to modify the horizontal and vertical displacement between the
stack nodes. The user can also disable or enable the *Collapse Stack Layout*.

**Tree Layout Strategy**

Trees are a common data structure in software programs. HDPV supports the application of a tree layout to a set of nodes. The user can apply a tree layout to a set of nodes rooted at a particular node by middle-clicking on the node which arranges the subtree from the selected root in the form of a tree. The nodes of the tree are highlighted to provide a visual cue to the user. Any new nodes that are added to the existing tree are automatically subjected to the tree layout rather than the default layout strategy. The tree layout can be de-activated.
through user interaction. When de-activated, the nodes that were part of the tree layout will be again subjected to the default layout policy.

Figure 3.22: Tree Layout applied to a set of heap nodes

Figure 3.22 shows the tree layout applied to a set of nodes.

**Layout Customization:** The tree layout supports two layout customizations:

1. The orientation of the tree can be selected. Supported orientations are ‘top-down’ and ‘left-right.’

2. The breadth and depth spacing of the tree nodes can be modified.

**Managing the Layout’s Stability:** The layout should be applied such that the stability of the visual graph is maintained [Cos03]. Stability refers to the property that when a layout is re-applied due to a change in the underlying graph structure, the graph layout should incur only minimal changes to the position of nodes to avoid user disorientation.

In our system, the tree layout is re-applied whenever a new node is added to the tree. To maintain the stability of the graph, HDPV animates the path taken by the node from its initial position to its new layout position. Changes in the position of the nodes caused by
layout customization, for instance when changing the orientation from left-right to top-down, are also animated.

**Visualization Persistence and Layout Considerations**

HDPV provides the flexibility to save the state of the program, including its visual attributes, to a persistent medium. The saved state can later be restored. Visualization persistence is required in the following cases:

1. Users may want to use HDPV to visualize a long running software program. In such a case, users need the capability to save the state of the program to a persistent medium and resume the visualization at a later point of time.

2. Users may want to play a set of visual actions repeatedly from a particular step in the visualization. In such cases, users could create a *visual bookmark.* For instance, consider a program that creates a list with a large number of nodes and performs a list manipulation operation. Users may be interested in visualizing this operation rather than the list creation sequence. If the visual state after the list creation is saved, users can load the saved visual state and continue the execution from that point.

Restoration of the visual state includes restoring the spatial location of the visual objects. During the course of the visualization, users may have laid out the objects according to their interest either by using any of the layout algorithms or by manual interaction with the visualization. When the visual objects are saved and restored later, users will expect the objects to be laid out exactly at locations where they were laid out when the visual state was saved. HDPV remembers the layout positions and other user defined changes and restores them to the same layout position.
3.4.3 Navigation Strategy

HDPV supports the *Zoom and Pan* and *Overview + Detail* navigational strategies.

**Zoom and Pan**

Regions of the visualization can be zoomed and/or panned. This zoom and Pan interaction strategy avoids the use of scroll bars to navigate to different areas of the visualization. The middle mouse wheel controls the amount of zooming. Currently, HDPV supports geometric zooming.

**Overview + Detail**

The *Overview* window provides an overview of the entire visualization. Users can obtain a overview of the entire visualization and navigate to regions of interest through the main visualization window. A cross-hair in the *Overview* window that controls the information displayed by the *Detail* view could be introduced for additional navigation.

3.4.4 Interaction Strategy

The goals of the interaction strategy are:

1. Users should be engaged cognitively in the visualization.

2. Scalability should be achieved by efficient use of screen space.

3. Layout strategies should enable ‘trial and error’ based exploration.

4. Interactive controls should be easy to use.
Table 3.2: Interaction controls

<table>
<thead>
<tr>
<th>Mouse Button Action</th>
<th>Left/Right Button</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Click</td>
<td>Left</td>
<td>Fix/Unfix (toggle) the node to/from a particular position</td>
</tr>
<tr>
<td>Single Click</td>
<td>Right</td>
<td>Zoom to fit the entire visualization to the screen</td>
</tr>
<tr>
<td>Double Click</td>
<td>Left</td>
<td>Cycle through the types assumed by the node</td>
</tr>
<tr>
<td>Middle Click</td>
<td>Nil</td>
<td>Apply Tree Layout from the selected node treated as ‘root’</td>
</tr>
<tr>
<td>Hover</td>
<td>Nil</td>
<td>Forward/Reverse Memory Reference Tracking</td>
</tr>
<tr>
<td>Wheel Movement</td>
<td>Nil</td>
<td>Control the zoom level</td>
</tr>
</tbody>
</table>

Table 3.2 shows the interactive capabilities that are provided through mouse controls.

**Fixing and Dragging**

HDPV allows floating nodes to be fixed to a particular position by *single-clicking* on the node. A visual cue is provided that a node is fixed by darkening the shade of the node. Users can also drag the nodes to a different location within the visualization. When the drag is released, the node is fixed to the dragged location. If the dragged node contains references, then the entire subtree is dragged along with the node.

**Memory Reference Tracking**

*Memory Reference Tracking* tracks points-to or reference relationships between memory blocks. Points-to relationship can be tracked interactively by hovering the mouse over a node which highlights the referenced objects. *Memory Reference Tracking* operates in two modes:

1. **Forward Reference Tracking**: In this mode, all memory blocks that are reachable directly or transitively from the currently selected memory region are highlighted.

2. **Reverse Reference Tracking**: In this mode, all memory blocks that refer to the currently
selected node are highlighted.

Figure 3.23: Tracking memory references: When Mouse is hovered over the `main()` function, all references from `main()` are highlighted.

Figure 3.23 shows an example. In the figure, as the user hovers the mouse over the stack node representing the `main` function, the nodes that are reachable from the stack node are highlighted.

**Multi-Type Representation Control**

In HDPV, users can apply different type representations to the same memory block by *double-clicking* on a particular node. Depending on the chosen type representation, the contents of the memory block is re-interpreted and re-displayed. This feature is useful when a memory block is type-cast to another type and also when visualizing a ‘union’ data type.
Figure 3.24: Node displayed in two different representations: Node is shown in two representations, as an array of characters and as an object of type `link_node`.

Figure 3.24 shows how a node can be alternately displayed as an array of characters and as an object of type `link_node`.

**Large Array Display Control**

The *Large Array Display* control allows the user to collapse and expand arrays. When using the *Drawn to Scale* approach, the nodes representing array regions would become huge for large arrays, cluttering the display space. To avoid this scenario, nodes displaying array regions can be collapsed and expanded through a collapse/expand box provided at the tail region of the array node. In collapsed mode, only the first 5 array values are displayed to the user. The entire set of array values can be viewed by clicking on the array expand/collapse box. A visual cue whether the array is collapsed or expanded is given through the expand/collapse mark displayed over the expand/collapse box.

Figure 3.25 and Figure 3.26 show how HDPV represents an array in collapsed and expanded views.

---

**Figure 3.25:** Array Representation: Collapsed mode

**Figure 3.26:** Array Representation: Expanded mode
Handling Large Program Structures by Collapsing and Expanding Nodes

HDPV is designed to handle programs that create large numbers of memory objects on the heap. However, as the number of nodes increases, interaction with the visualization may become difficult. By collapsing and expanding nodes, users can manage a large number of visual items in a limited display space. Collapsing and Expanding is an example of Details on Demand where unwanted details can be hidden from view and re-displayed when required. The user can collapse a node by clicking on the Collapse/Expand Control region of the node. Once collapsed, all nodes that are reachable from the current node through any memory reference will be hidden from the display. A visual cue is provided to the user that the nodes are hidden by drawing a thick boundary around the collapsed node. The collapsed node can be expanded again by clicking on the same Collapse/Expand Control region.

Figure 3.27: Node in expanded mode: Node containing references to a set of nodes shown in expanded mode

Figure 3.27 and 3.28 show the collapse/expand capability of HDPV. Figure 3.27 shows a node with references to other nodes. The figure also shows a Collapse/Expand button at

61
Figure 3.28: Node in collapsed mode: The thick boundary around the node signifies that the set of nodes emanating from it is hidden in the right top corner of the node. Figure 3.28 shows the node after collapsing the references emanating from that node. The thick boundary around the node signifies that the node is collapsed.

Figure 3.29 shows HDPV visualizing a program that creates 100 nodes of a linked list each holding a pointer to a binary tree containing 10 nodes.

Figure 3.29: Handling large graphs - Initial Snapshot: HDPV displaying a large heap structure
Figure 3.30 shows the system after the user has interactively collapsed some nodes of the graph. The shown graph structure was produced after collapsing around 200 nodes of the graph.

![Image of the system after node collapse]

**Figure 3.30: Handling large graphs - Applying Node Collapse Control: Snapshot of the system after interactively collapsing the nodes of the graph**

**Tool Tip Capability**

The name of the record types is displayed as part of the node; for activation record types, the name of the function, its parameters and return types are displayed. Additional information about the types cannot be displayed in the limited amount of space as it would clutter the display. Therefore, HDPV provides tooltips which display detailed information about the type of a node when the user hovers the mouse over the node. For activation records, information about parameters, local variables, and return type are displayed. For other record types, name and the fields of the type are displayed.
Application of Custom Layouts

Users can apply different layouts to the visualization through interactive controls on a ‘trial and error’ basis. Currently, HDPV supports Collapse Stack layout, Force Directed layout, Global layout and Tree layout which are discussed in Section 4.2.3. The *Collapse Stack Layout* can be activated through a separate layout control panel. A *Tree Layout* can be activated by *middle-clicking* on a particular node which will become the root of the tree. Once the layouts are applied, the user is provided with options to customize the layouts.

Figure 3.31 shows a snapshot of a heap structure with tree layouts applied to different portions of the heap structure. Through the layout controls, one of the tree structures has been customized to be oriented *left-right*, while the other trees are laid out in a *top-down* fashion.
Figure 3.31: Application of Tree Layout: Snapshot showing tree layouts applied to different parts of the heap structure. The layouts have also been customized - the highlighted tree structure is oriented left-right whereas the other tree structures are shown in default orientation.
Chapter 4

Implementation

This chapter discusses the implementation details of HDPV. Section 4.1 discusses the implementation of the Monitor and Section 4.2 discusses the implementation of the Visualizer.

4.1 Monitor Implementation

The Monitor is implemented using the Top infrastructure as a plugin for the Pin instrumentation environment. The Monitor is built around two modules. The Type Information Extractor collects type information from the C/C++ executable and the Runtime State Monitor monitors the program for runtime state changes. The Type Information Extractor is invoked on loading the C/C++ binary. It collects compile time information about the types used in the program and communicates this information to the visualizer. As the program starts executing, the Runtime State Monitor traces the program to extract runtime state information.

Section 4.1.1 explains the DWARF debugging information format used by Type Information Extractor. Section 4.1.2 explains the implementation of the Type Information Extractor.
and Section 4.1.3 describes the implementation of the Runtime State Monitor.

4.1.1 The DWARF Debugging Information Format

DWARF is a debugging format used by many debuggers to support source level debugging. The Type Information Extractor uses DWARF information to collect compile-time type information.

Programs contain data types whose information must be available to the Visualizer. Program visualization systems which operate directly on binaries must infer the type information from the debugging information present in the executable.

Native binaries include sections that describe the code segment, data segment, and stack segment of a program. In addition to these sections, an executable includes a Debugging Section if the ‘-g’ flag is passed to the compiler. This section contains debugging information in a format known as DWARF Debugging Format.

DWARF information contains debugging entries to define a low-level representation of elements contained in the source program. Each debugging entry consists of an identifying tag containing a set of attributes. The tag identifies the class to which the entry belongs and the attributes define the characteristics of the entry using name-value pairs.

A sample DWARF debugging entry is shown below:

```
<2><a6>: Abbrev Number: 3 (DW_TAG_formal_parameter)
  <a7> DW_AT_name : param2
  <a9> DW_AT_decl_file : 1
  <aa> DW_AT_decl_line : 3
  <ab> DW_AT_type : <dc>
  <af> DW_AT_location : 2 byte block: 91 4 (DW_OP_fbreg: 4)
```

Here, DW_TAG_formal_parameter is a tag denoting that the debugging entry encodes in-
formation about a formal parameter. DW_AT_name, DW_AT_decl_file, DW_AT_decl_line, DW_AT_type and DW_AT_location are attributes encoded as name-value pairs. DW_AT_name denotes the name of the attribute, DW_AT_decl_file denotes the file in which it occurs, DW_AT_decl_line denotes the line number in the file, DW_AT_type denotes the type of attribute and DW_AT_location denotes the location information which could be location expression or a location list which is explained below.

The Monitor needs location information to trace the variables used in the program. The DWARF debugging format provides a convenient way to specify the location of program variables, find the bounds of static arrays, and determine the base address of a function’s stack frame. Location information is provided as location descriptions in one of the following ways:

- Location Expressions, which provide a language independent representation of addressing rules that describe objects whose location does not change during their lifetime.

- Location Lists, which describe objects that have limited lifetime or whose location changes during their lifetime.

Whereas variables are located through location expressions or location lists, activation records are located through a Canonical Frame Address. A call-frame denotes an area of memory allocated on the stack for a subroutine. A call-frame is associated with an address known as Canonical Frame Address or CFA.
4.1.2 Type Information Extractor

Use of fjalar DWARF Reading Library

The Type Information Extractor heavily re-uses the functionality provided by the fjalar dynamic analysis framework. Fjalar [Guo06] is a dynamic analysis framework for constructing dynamic analysis tools for programs written in C/C++. fjalar provides both compile time and runtime services to tools that are built on top of it.

fjalar provides compile-time information about data structures, variables and functions. In order to obtain this information, fjalar parses the debugging information, filters out redundant information, and sorts out the information obtained into suitable entries for types, variables and functions. It also allows tools to traverse through a program’s data structures at runtime and perform suitable operations by inspecting them. To perform analysis at runtime, fjalar uses the Valgrind binary instrumentation framework.

We used only the compile time services provided by fjalar for our purpose because fjalar runtime services’s use of Valgrind binary instrumentation framework is incompatible with Pin. We separated out those parts of the framework that provide compile time information about types, variables and functions into a DWARF reading library that parses the binary and represents the extracted information in suitable data structures.

The following steps are performed by the DWARF library to collect the debugging information:

1. The data structures required for storing the type information are initialized.

2. The program binary is read and its DWARF information is parsed. The parsed information is stored in a DWARF specific data structure.

3. The information available in the DWARF specific data structure is processed to seg-
regate them into fjalar specific data structures. These data structures store the data collected in separate containers that facilitate iteration over different data types and function definitions.

Once the above steps are performed, the required information is in the following fjalar specific data structures:

**TypeEntry:** This data structure holds required information about a particular data type defined in the program. It contains the name of the type, size, and a list of field members for this type. Additionally, if the type is a C++ class type, it stores the list of constructors, member functions and destructor defined in the program.

**FunctionEntry:** This data structure contains information about all functions defined in the program. It holds the name of each function, the list of formal parameters, and the list of local variables defined in the function.

**VariableEntry:** This data structure holds specific information about all variables used in the program. It holds the name of each variable, the type of the variable, which is a reference to TypeEntry, the location of the variable within the program, and flags to specify if the variable is a global variable, static variable, array, or reference variable.

**Data Types Processing**

The type information must be announced to the visualizer to render the memory blocks using suitable types. For each type defined in the program, the filename, size, offset and field type are extracted from the TypeEntry data structure and encoded into XML messages that are sent to the visualizer.
Constructing Padded Regions for a Type: The compiler may introduce padding when packing the fields contained in a data type to align the types according to the alignment rules of the architecture. Consider the following type definition:

```c
struct temp {
    char c;
    int b;
}
```

In the above example, the compiler introduces 3 bytes of padding after the variable `c` to align the address of `temp.b` on a word boundary. In such a case, the Monitor introduces `Padding` fields to make up those padded memory regions.

The XML message announcing a padded type definition is shown in Figure 4.1.

```
<typedef name="temp" size="8">
    <field name="c" size="1" offset="0">
        <char/>
    </field>
    <field name="Padding" size="3" offset="1">
        <uint32/>
    </field>
    <field name="b" size="4" offset="4">
        <int32/>
    </field>
</typedef>
```

Figure 4.1: XML message for a padded type

The Monitor calculates if the structure contains a padding by comparing the size of the field with the difference of the offsets at which successive field members are located. If the size of a field is smaller than the difference in the offset, the Monitor introduces a ‘Padding’ field to compensate.
Activation Record Types Processing

Activation record types are treated similar to other data types. The activation record of a function consists of a name, a set of formal parameters, and local variables, which are treated as fields of a data type. Each function is stored in a function map so that the function’s activation record type can be announced when the function is first entered during the execution of the program.

4.1.3 Runtime State Monitor

The Runtime state Monitor traces the program to collect runtime information. The Monitor uses a Interval Tree data structure to manage the memory blocks created in the program.

Interval Tree

A program’s state contains memory blocks for function activation records, global variables, and object allocated on the heap. The Monitor uses the MemoryAccess event provided by the Top framework for monitoring accesses to memory. Events are triggered when a memory access is detected within any of the registered addresses. We need an efficient way to identify the memory block that contains the memory access. The Interval Tree data structure contains all memory intervals used in the program. These memory intervals are maintained as a AVL tree data structure. As variable assignments occur in the program, this data structure is consulted to retrieve the block to which the write access occurred.

The Interval tree is illustrated in Figure 4.2. The data structure supports the following methods:

insert: The ‘insert’ method inserts an interval into the AVL tree data structure.
Figure 4.2: Interval Type Illustration: The figures within the objects represent sample memory addresses

**find**: The ‘find’ method retrieves the interval that contains the given memory block. This operation involves traversing the AVL tree data structure and finding the right interval.

**remove**: The ‘remove’ method removes an entry from the tree data structure.

As the program is executed in the *Pin* instrumentation environment, the Runtime State Monitor listens for *Function Call*, *Function Return* and *Memory Access* events from the program.

**Function Entry**

The Monitor tracks the entry of a function through the receipt of a *Function Call Event* from the *Top* framework. When a function is entered, the Monitor announces the entry of the function to the Visualizer. It constructs a call frame representing the function, registers for memory access events to track memory writes and sends a XML message signaling the allocation of a new stack frame to the Visualizer.
**Extracting Call-frame Information:** The System V Application Binary Interface (ABI) defines the procedure calling conventions, which include stack frame layout, register usage, and parameter passing conventions. The compiler has the liberty to freely lay out those elements whose position is not specified by the ABI specification. For instance, the compiler may allocate extra locations for saving intermediate values computed within the function. It may also allocate locations for Register Spilling. The challenge is to construct the exact stack frame layout used by the function without knowing the compiler’s strategy. Our approach is similar to that of the debuggers that use the debugging information present in the executable to identify the function’s stack frame.

An activation record, according to the System V ABI, consists of formal parameters, a set of locations to store the callee saved register(s) and return address, a set of local variables and some extra regions allocated by the compiler. As soon as the function is entered, the stack pointer will be adjusted to accommodate space for the local variables. The total size of the function is given by the space between the first formal parameter and the last local variable pushed onto the stack. However, when a Function Call Event is received by the tool, the stack pointer adjustment for local variables has not yet occurred, therefore, the Monitor has to calculate the total size of the function from the DWARF information present in the executable. Since all the variables within the function carry known offsets within the activation record, the size of the activation record can be calculated as the sum of the maximum value of all local variable offsets plus the maximum value of all formal parameter offsets plus the size of the last formal parameter.

To understand how the Monitor computes the stack frame layout for a function, consider the following code sample:
```c
void f(int param1, float param2) {
    int local1;
    double local2;
}
```

Table 4.1 shows the list of formal parameters and local variables and their corresponding size and location extracted from DWARF information.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Offset</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>param1</td>
<td>Formal Parameter</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>param2</td>
<td>Formal Parameter</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>local1</td>
<td>Local Variable</td>
<td>-20</td>
<td>4</td>
</tr>
<tr>
<td>local2</td>
<td>Local Variable</td>
<td>-16</td>
<td>8</td>
</tr>
</tbody>
</table>

The stack size is calculated as the maximum offset of local variables, in this case 20, plus the maximum offset of formal parameters, which is 4, added to the size of the last formal parameter b, which is 4, totaling to 28. Figure 4.3 illustrates the stack frame layout for this example.

The Monitor determines if the function activation record layout has already been announced. If not, the activation record type of the function is announced through a `typedef` message. Figure 4.4 shows the XML message sent.

Once the type information has been communicated to the visualizer, a XML memory allocation message is generated for the memory allocated for the call frame representing the function’s activation record.

**Special Handling for main function:** The `main()` function is usually the first user defined function invoked by any C/C++ program. This function differs from other functions in the following ways:

1. `main` takes command line arguments. Information about the command line arguments,
Figure 4.3: Stack frame layout for a sample function: Stack frame and its corresponding offsets at which variables and formal parameters are located are shown which are provided to the program when it is invoked, is available through \texttt{main}'s formal parameters. The first parameter denotes the number of arguments, the second parameter contains a reference to the list of command line parameters and the third parameter contains a reference to the environment variables.

2. Gcc performs stack alignment operations when the main function is entered. Gcc aligns the stack to a power of 2; 16 by default. Therefore, the activation record for \texttt{main} must account for this alignment.

The size of the stack frame for \texttt{main} function is the sum of the size of the following items:

- Command line arguments (if defined in the program)
- Space allocated due to stack alignment
- Saved registers such as EBP, EIP, ECX etc.
Figure 4.4: XML message used for activation record announcement

- Local variables declared within the main function

Figure 4.5 illustrates the stack frame layout of main.

The parameters are inspected to see if the user declared the parameters to receive command line arguments. If so, these parameters are included in the call-frame. Next, the extra space allocated due to the stack boundary alignment performed by the compiler is identified and included. Then the intermediate space that exists between the memory address and local variables is adjusted with space for saved registers. Finally, the local variables are considered.

MemoryAccess events are registered with the Top framework for the memory region representing the function stack so that any assignments to memory regions that occur within the function will be passed to the Monitor. Also, a new entry is created in the Interval Tree.
Figure 4.5: Stack frame layout of the main() function for the memory region representing the function’s activation record.

**Function Exit**

When the function returns, the Monitor notifies the visualizer that the function has exited and all associated shadow state for the function must be cleaned up. The Monitor announces the exit of a function by generating a `memfree` message that contains the address and size of memory to be deallocated. The `MemoryAccess` events that were previously registered with the Top framework are de-registered and the entry for the region in the Interval Tree is removed.
Assignments

Assignments in the program are tracked by the Monitor through the receipt of MemoryAccess events from the Top framework. This event includes the address of memory access, along with the type (which can be memory read or memory write) and the size of memory access.

Assignments are communicated to the visualizer through Putfield messages. A Putfield message carries an address, an offset, and a value pertaining to a particular field type. The ‘offset’ is calculated as the offset of the address being accessed from the base of the memory region. The memory region that contains this address is obtained by querying the Interval Tree. The offset is calculated as $\text{offset} = \text{Access Address} - \text{Base Address}(X)$. The region’s base address is included in the Putfield message to allow the Visualizer to identify the corresponding shadow region.

Next, the assigned value must be calculated. The value depends on the size of the memory access because the size of the memory access determines the type of the variable that was assigned in the program source. For instance, $\text{char } a = 'h';$ results in a movb instruction at the assembly level whose memory write size is 1, whereas $\text{int } b = 500;$ results in a movl instruction whose memory write size is 4. Depending on the size of the write, a byte, word, or double word is read from memory and the value at that location is interpreted according to its size. The Putfield message uses one of the four data types, uint8, uint16, uint32, or uint64.

Consider a assignment:

\[
\text{int } a = 5;
\]

If \(a\) is a local variable and the offset of ‘\(a\)’ within the function stack is 4, then assignment of 5 to variable ‘\(a\)’ will result in the XML message to be generated as shown in Figure 4.6.

Here in this example, ‘address’ refers to the base of the memory region, which in this case is
a stack frame.

Heap Memory Allocation

Memory allocation events are detected by tracking calls to memory allocation functions. During the initialization phase of the tool, memory allocation functions such as `malloc` or `calloc` are registered with the Top framework. Consequently, when memory is allocated from the heap though a call to any of these functions, the Monitor is notified by the framework.

The Monitor learns the number of bytes allocated by inspecting the parameter passed to the `malloc` function call. It obtains the address of the memory allocated by inspecting the return value of the call. The Monitor then adds the memory block allocated into the Interval Tree, registers MemoryAccess event with Top and sends a Memalloc message with the address and size of the memory allocated.

The field type of the newly allocated memory region is always initially announced as an array of characters because, we do not yet know the type eventually associated with that piece of memory. When this memory block is assigned to a pointer of a particular type, the type information becomes known to the Monitor. Even if the type information is known to the Monitor when the memory is assigned to a pointer, the Monitor treats assignments to pointers as if they were integer assignments. The visualizer infers the type of the memory block based on the type of variable to which it is assigned. This technique allows us to

```xml
<putfield address="bf8f1928" offset="4">
  <int32 value="5"/>
</putfield>
```

Figure 4.6: XML message generated for Putfield announcement
handle compilers that emit integer instructions to move floating point values.

**Heap Memory Deallocation**

Memory de-allocation events are detected by tracing calls to deallocation functions such as `free`. When a `free` call is detected, the Monitor collects the address passed to `free` and retrieves the memory region from the *interval tree*. The size of the memory is given by the bounds of the interval object. Once the size and the address is obtained, the Monitor removes the entry from *Interval Tree*, de-registers *MemoryAccess* event with *Top* and generates a *Memfree* message.

**Source Line Information Gathering**

Source line information can be gathered either when the program execution advances from one source line to the next, or when the state of the program undergoes a change. We use the second approach because successive source code lines may not cause changes to the program state. We wanted the source line information to be sent only when there is a visual state change to be reflected on the Visualizer side.

*Pin* exposes an API named `PIN_FindLineFileByAddress` to extract file name and line number of a given address in the program. This API requires the source program to be compiled with debugging information by passing ‘-g’ flag to the compiler. The API takes the current value held by the Instruction Pointer to obtain the line number information. Line number information is communicated to the Visualizer through a *location* message.

4.1.4 Assumptions and Limitations

The Monitor makes the following assumptions when tracing the program:
1. The Monitor assumes that the layout of the stack frame does not change during the course of the function. This assumption is violated if a call to *alloca* is made within the function. *alloca* allocates memory on the stack by moving the stack pointer. We currently do not recompute the stack frame layout once a function is entered. Therefore, the visualizer will not show the change in the stack frame layout. We can send a type definition for the new activation record and instruct the Visualizer to use the new type definition for the stack node using a *showas* message. However, functions such as *alloca* are used rarely in C/C++ programs.

2. The variables are assumed to reside in fixed positions within the stack. This assumption is true for programs compiled without compiler optimizations. If the code is optimized, then the variables might reside in stack until a particular region in the program and later might be available in a register. When these variables are eventually written to memory, the visualizer will visualize the assignment.

3. All local variables are assumed to reside in distinct memory regions within the stack; there is no memory region reuse. This assumption might not be true for cases in which two variables that reside in distinct code blocks within a function share the same stack space. Consider the following code sample:

```c
void func() {
    {
        int a[10];
    } {
        int b[10];
    }
}
```

In this case, the compiler might choose to allocate same stack location for the array `a` as well as `b`. As the control switches from one block to another, the layout of the stack
frame should change to represent variable \( b \) occupying the region previously occupied by \( a \). In our implementation, however, whenever updates to \( b \) occur in the program, the visualizer would show them as updates to \( a \).

**Monitoring Uninstrumented Program Regions:** Some parts of the program, such as library functions, are not subjected to monitoring because of the following reasons:

1. Since we do not know in advance what library functions will be invoked in the program, we cannot register for *Function Call / Function Return* events with the *Top* framework for tracking those calls. Therefore, we cannot detect when such library functions are called from the program. This issue could be handled by instrumenting all library functions proactively. However, the monitoring process would then incur additional overhead induced by these additional function calls.

2. The library functions are dynamically linked with the executable. Therefore, we may not have any debugging information for those program segments. This issue could be handled by compiling the program statically with the debugging version of the library.

### 4.2 Visualizer Implementation

The Visualizer consists of a graphical frontend that visualizes the program state and a shadow state backend that maintains the program state information. As the user steps through the visualization, the shadow state backend reads XML messages from the monitor and updates the program state information it maintains. Section 4.2.1 describes the implementation of our *ShadowState* data structure. When program state changes cause changes to the visual state, the backend redraws or updates the visual objects.

The Visualizer is implemented using the prefuse visualization toolkit [HCL05]. Prefuse
provides Java classes that can be used for building interactive information visualization applications. Renderers in prefuse are used for drawing visual items to a graphics context. We customized the default renderers extensively, which is described in section 4.2.2. Whereas most layout algorithms produced by prefuse did not need modification, we redesigned the *NodeLinkTreeLayout* module provided by prefuse to suit our requirements. Section 4.2.3 describes our design of *Layout Groups* to manage the different layouts applied to a visualization.

## 4.2.1 Shadow State Implementation

The Visualizer uses the *ShadowState* data structure to maintain the program state information. This data structure is needed because the communication is unidirectional from the monitor to the Visualizer, so the visualizer cannot refer to the Monitor for program state information. The shadow state uses prefuse’s *Graph* class to represent the canonical state of the program. As the Visualizer receives the program state information from the monitor, this data structure is updated.

**MemoryBlock:** The *MemoryBlock* data structure represents the memory blocks created in the program. It consists of two components, a one dimensional array of bytes that stores the values contained in the memory block and a bit field to represent those bytes that are initialized. The memory is maintained as an one dimensional byte array to facilitate different type representations.

**Initialization of Shadow State**

The *ShadowState* data structure is initialized by creating a state graph that consists of a set of nodes and edges backed by tables that hold the data. The state graph is directed. This
Prefuse organizes visualization data as ‘Tables’ whose columns are described by a schema. Since a graph consists of a set of nodes and edges, prefuse requires creating node and edge schemas. The ShadowState data structure creates schemas with the fields required for maintaining the state information for each node, which includes the name of the type, the size of the memory block represented by the node and the kind of memory block, which can be stack, heap or global. The schema also contains a reference to a MemoryBlock object, and a list of strings that caches labels used by the Renderer.

The Edge Schema consists of fields that represent the source offset and destination offset of the memory blocks.

**Processing XML Messages**

As the XML messages are received by the visualizer from the Monitor, the ShadowState data structure is updated according to the type of the message received by the Visualizer.

**Memalloc:** A memalloc message signals new memory blocks created in the program. The visualizer creates a new graph node and adds it to the state graph. The type, address, size, and kind fields of the node are assigned with values obtained from the message.

**Memfree:** A Memfree message signals deallocation of an existing piece of memory. The node that corresponds to the memory region to be freed is identified and removed from the graph. Along with the node, all edges incident on the nodes are also removed.

**Typedef:** The shadow state maintains the type information received from the typedef message by storing the type object in a hash map indexed by typename. This data structure is
consulted whenever references to types are made.

**PutField:** On receipt of a *putfield* message, the *MemoryBlock* object corresponding to the node is updated with the value of the message at the corresponding offset. If the field type represents a pointer type, an edge must be created that connects the source node identified by the address field of the *putfield* message with the destination node identified by the value field of the message.

Edges can originate and end at any offset within a memory region. Therefore, the edges of the shadow state graph must maintain the source and destination offsets.

Once the shadow state object is updated, the node corresponding to the memory region must be updated to reflect the changed value. This update is communicated to the listeners responsible for relaying the updates to the renderers.

**Location:** *Source Code Highlighting* is implemented through the *GVim* editor. The location information is communicated to the editor to highlight the corresponding source line in execution.

### 4.2.2 Custom Renderers

Prefuse renderers draw the visual items to the graphic context. The methods of prefuse’s *LabelRenderer* class render the nodes of the graph and *EdgeRenderer*’s methods render the edges of the graph. Prefuse enables custom renderers to be plugged in to modify the default rendering policy. Though the default renderers provided the basic functionality of drawing simple nodes and edges, they lacked the ability to support the customizations required for our Visualizer. Therefore, we subclassed the default renderers to implement the functionality we needed.
The reasons for designing a custom node renderer are:

1. The size of nodes, according to our *Drawn to Scale* approach, must be drawn based on the size of the memory region they represent. However, the default renderer decides the size of the node based on the length of the node label.

2. The default renderer draws the same shape for every node in the graph. However, we require the ability to display each node differently according to the type it represents.

3. In addition to the value, the nodes must display the type name associated with the memory block. The default renderer lacks this capability.

4. Nodes must display the value of all fields contained within the particular type. However, the default renderer can display only one value per item.

The Visualizer runs the renderer modules whenever the layout is computed. Layout algorithms such as *Force Directed Layout* are continuously recomputed, requiring the renderers to be as efficient as possible. Therefore, the width and height dimensions of the rendering items, which depend on the node’s type, are pre-computed to avoid having to recompute them repeatedly. The pre-computed information is cached in a `RenderInfo` object, which is stored in a hashmap indexed by type.

The `RenderInfo` object is constructed for every type encountered in the system. Every node contains a reference to a `RenderInfo` object which is consulted by the rendering code. Since multiple nodes can correspond to the same type, multiple nodes contain a reference to the same `RenderInfo` object.

The `RenderInfo` object is constructed by iterating over the fields of the type, calculating the width and height required for drawing all fields contained in the type. This object also stores the orientation of the node, which can be *Horizontal* or *Vertical*. Special case
handling is required for types that do not have any fields within them. Even though the Monitor announces them with a size of zero, they must be represented in the visualization. For such types, a default width and height are computed and stored.

The canonical state model gives the flexibility to store multiple types associated with the same memory object. The renderer consults the current type representation assumed by the node when computing the RenderInfo object. The types associated with a given node can change in two ways, either when the user explicitly requests that the node should be viewed in a different type or when the Monitor hints to the visualizer that the type be changed through a ShowAs message. If the renderer finds that the type associated with a particular node has changed due to user interaction, it removes the old association of type with the node, reassigns it to the new type and recomputes the dimensions of the shape representing the new type.

The custom node renderer considers language-specific naming conventions. C/C++ pointers variable are preceded by a `*` when being displayed, whereas the `*` is omitted for Java reference variables. Because both variable types are represented alike by the intermediate XML representation, the user must supply the preferred language when invoking the visualization.

The default edge renderer had to be modified to take source and destination offsets into account. The default edge renderer positions the lines drawn for edges such that they start and end at the center of the source and destination nodes. Since references can originate at any offset within the memory region and point to any location within another memory region, our custom renderer inspects the source and destination offset fields and relocates the edge such that it originates from the source offset within the source node and ends at the destination offset within the target node.
4.2.3 Layout Groups

Prefuse provides a set of graph layout algorithms including Force Directed Layout, Node Link Tree Layout and Radial Layout. The Force Directed and Collapse Stack layout operates over a set of visible items identified by a group. The Node Link Tree layout requires a prefuse graph instance, constructs a spanning tree starting from a root node, and lays out the nodes of the spanning tree using a NodeLinkLayout.

To address this non-uniformity, we designed Layout Groups. Layout groups encapsulate layout specific properties and their implementation details, allowing new layout strategies to be added easily. Every node in the visualization is part of a Layout Group. On user interaction, nodes may move from one layout group to another.

Implementation

The key data structures involved in our layout group implementation are LayoutGroup and LayoutGroupManager.

**LayoutGroup:** This interface encapsulates the layout algorithms provided by prefuse. It defines a set of methods such as run(), createControlPanel() and highlightContainingNodes() which must be implemented by the classes that inherit this interface.

**LayoutGroupManager:** This class manages the list of LayoutGroup objects.

Updates to Layout Groups are implemented through the Observer design pattern [EGV94]. The LayoutGroupManager maintains the list of all layout groups that are part of the visualization. It listens for updates to the underlying state graph, which is managed by ShadowState object. Updates include the creation of a new node, removal of an existing node,
Table 4.2: Layout Groups and their function

<table>
<thead>
<tr>
<th>Layout Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CollapseStack</td>
<td>Collection of stack nodes when CollapseStackLayout is activated</td>
</tr>
<tr>
<td>Global</td>
<td>Collection of global nodes that satisfy a particular filtering criteria</td>
</tr>
<tr>
<td>Force</td>
<td>Collection of heap nodes that are subjected to Force simulation</td>
</tr>
<tr>
<td>Tree</td>
<td>Set of nodes and edges for which TreeLayout is applied that arranges the set of nodes as a tree structure</td>
</tr>
</tbody>
</table>

creation of a new edge, and removal of an existing edge. Once the LayoutGroupManager receives the updates, it relays the updates to all the layout groups registered with the manager. Individual layout groups decide if the layout must be re-run for the group of nodes they manage.

Currently, HDPV supports the Collapse Stack, Global, Force, and Tree layout groups as shown in Table 4.2.

The Collapse Stack Layout group encapsulates the CollapseStackLayout instance that collapses a set of stack nodes and lays them out as a tile. The group maintains the list of stack nodes so that the layout can be activated and de-activated as desired. The layout group provides controls for changing the horizontal and vertical spacing between the stack nodes.

The Force layout group encapsulates a prefuse ForceDirectedLayout instance that lays out the floating nodes of the graph based on the force simulation as described in Section 2.1.2. Initially, all heap nodes belong to this layout group.

A Tree Layout group encapsulates a prefuse NodeLinkTreeLayout instance that lays out a set of nodes in the form of a tree. Since a NodeLinkTreeLayout acts on a set of nodes that are part of a subtree, the subtree’s root can be used to identify a layout group instance.

The Global layout group manages the set of global nodes and displays those global objects that satisfy a particular filtering criteria. For instance, static objects contained in Java
system classes that would otherwise clutter the screen can be hidden from the user.

4.2.4 Node Link Tree Layout

Prefuse provides an implementation of the \textit{NodeLinkTreeLayout} layout algorithm. This layout constructs a spanning tree over a graph and applies a tree layout algorithm \cite{BJL02}. Since the provided algorithm is unaware of the semantics of the data structure over which it operates, it can lead to layouts that do not reflect the type information or semantics of the program being visualized.

There are at least two cases when the default node layout algorithm does not create a meaningful layout.

1. Consider the layout produced by the default tree layout for a binary tree as shown in Figure~\ref{fig:prefuse.tree.layout}. The left and the right subtrees of one of the nodes are interchanged because the default tree layout lays out the nodes of the tree based on the order in which it encounters the edges of the node. In this case, the first edge was created by assigning the ‘right’ pointer, followed by assigning the ‘left’ pointer, leading to such a layout.

Figure 4.7: Layout produced by prefuse’s Tree Layout - 1: The left and the right subtrees are interchanged due to the order in which edges are inserted into the graph.
2. There may be null pointers in some nodes’ fields which are not represented by any edges. Since the default layout tries to pack all child nodes as closely as possible, the nodes will not be positioned appropriately. Figure 4.8 shows one such layout in which a right child is drawn centered rather than offset to the right.

Figure 4.8: Layout produced by prefuse’s Tree Layout - 2: The right subtree is oriented towards the left due to the missing left subtree

To handle these shortcomings, we redesigned the NodeLinkTreelayout algorithm provided by prefuse to meet our requirements. We redesigned the algorithm such that the original layout algorithm is largely untouched, but at the same time introduced our changes in a clean and maintainable way using the Decorator design pattern \[EGV94\]. We wrapped the default prefuse visual objects into a set of classes that determine how the layout position must be calculated.

The prefuse NodeLinkTreelayout algorithm iterates over the list of children of a particular node in order to decide the position of the nodes. Whenever node is processed for layout, the decorator creates either a ItemWrapper or a GhostItemWrapper instance, depending on the type of the node, whether the node represents a null pointer or an actual reference. The ItemWrapper wraps an actual node item instance that represents a node in the visual graph, whereas the GhostItemWrapper wraps a ghost node. The ghost node influences the layout algorithm such that the actual nodes are laid out taking into account the space that would be taken up if an actual node were present at that position. However, once the layout
positions are calculated, the ghost nodes are not rendered. This strategy maintains the correct ordering of the edges as well as assigning semantically meaningful layout positions for the actual nodes in the graph.
Chapter 5

Evaluation

This chapter describes the evaluation of HDPV. Section 5.1 illustrates the applicability of HDPV through a set of use cases. Section 5.2 evaluates the performance of HDPV. Section 5.3 discusses current limitations of HDPV.

5.1 Example Use Cases

HDPV finds its application in many areas. We categorized these areas in 3 groups:

- Understanding basic concepts in Computer Science
- Demonstrating algorithm implementations
- Visual debugging

In the following subsections, we provide a set of use cases in each of the above mentioned groups.
5.1.1 Understanding Basic Concepts in Computer Science

HDPV can help a student understand several basic concepts in Computer Science.

Understanding Recursion

The concept of recursion is better explained using conceptual models than abstract models [WDB98]. HDPV provides one such conceptual model that illustrates how recursion is implemented in programming languages. Computing the factorial of a number is a classic example of a problem that uses recursion. The source code for a C function that calculates the factorial is shown in Figure 5.1. Figure 5.2 shows how HDPV visualizes a recursive program.

```
1 int factorial(int n) {
2    int result;
3    if (n == 1)
4        result = 1;
5    else
6        result = n * factorial(n - 1);
7    return result;
8 }
```

Figure 5.1: Code Sample: Factorial program

In this example, as the recursive program progresses, new stack frames are created until the end condition for the recursion is reached. As the recursion reaches the termination condition, the stack unwinds and the results appear as shown in Figure 5.3. Failure to include the end condition for recursion would be shown by an indefinite increase in the number of activation records. In this example, students see how recursion is implemented in programming languages. They can recognize that, although the programmer declared n and result only once, a fresh copy of those variables is allocated on the stack for every recursive
Understanding ‘this’

In languages such as C++ and Java, the ‘this’ pointer is implemented as an invisible argument passed to every non-static method of a class. HDPV visualizes the ‘this’ pointer by showing ‘this’ as one of the arguments to every non-static method call.

Consider a program that creates a binary tree and uses a traverse() method to traverse its nodes. The code snippet for traverse() is shown in Figure 5.4. The ‘this’ pointer in the traverse() method holds a reference to the current node that is being traversed.

Figure 5.5 how HDPV visualizes the ‘this’ pointer which holds a reference to the current
Figure 5.3: Factorial program - Intermediate results on stack

```cpp
class Node {
    Node *left, *right;

    public:
    Node(Node *left, Node *right) {
        this->left = left;
        this->right = right;
    }

    Node() {
        this->left = this->right = NULL;
    }

    void traverse() {
        if (this->left)
            this->left->traverse();
        if (this->right)
            this->right->traverse();
    }
};
```

Figure 5.4: Code Sample: Understanding 'this'

97
object. As the `traverse()` method traverses each node in the tree, the object that is currently in scope is held by the ‘this’ pointer. This fact can be ascertained by using the *Forward Reference Tracking* feature that allows users to hover the mouse over the stack frame and observe how the nodes pointed to by ‘this’ are highlighted.

Figure 5.5: Understanding ‘this’: the ‘this’ pointer holds a reference to the current object. The set of objects reachable through the ‘this’ pointer are highlighted when the user hovers over the stack node that contains ‘this’

**Object Allocation in Programming Languages**

Different languages use different syntaxes to allocate objects. Consider the segments of code shown in Figure 5.6 that create objects in C++ and Java:
Account acc; //C++: create local object
Account *acc = new Account(); //C++: create zero-initialized object in heap
Account *acc = new Account; //C++: create un-initialized object in heap
Account acc = new Account(); //Java: Create zero-initialized object in heap

Figure 5.6: Code Sample: Object Creation

Account acc: Creates a local object on the stack. This construct is used in C++. Though Java supports a similar looking construct, Java’s version creates an uninitialized reference, rather than an object on the stack.

Account *acc = new Account(): The object is allocated on the heap and is held by a pointer acc that is allocated on the stack. Because of the use of (), the C++ object is zero initialized.

Account *acc = new Account: The object is allocated on the heap and is held by a pointer acc that is allocated in stack. However, the object’s fields are not initialized. This construct is used in C++.

Account acc = new Account(): A Java construct that is used to create a new object. The object is initialized by its constructor.

Figure 5.7 shows how HDPV demonstrates these four constructs visually. This example demonstrates how HDPV can show how similar concepts in different languages correspond to each other. Subfigure (c) denotes an object created in C++ whereas subfigure (d) denotes an object created in Java. Both representations look similar to the users except that, in C++, acc is a pointer denoted by a ‘*’, while in Java, a reference variable is shown without a ‘*.’ This uniform representation helps users clearly understand different implementations of the same concept.
Understanding Aliasing

HDPV can visualize aliasing relationships.

The code sample shown in Figure 5.8 creates a linked list that is held by the ‘start’ pointer located in main. This function has ownership of the list. Subsequently, the ‘start’ pointer is passed to the detect_cycle function which detects a cycle in the linked list. In detect_cycle, an alias to the list is created. Users can visualize multiple aliases that refer to the list by using HDPV’s Reverse Reference Tracking feature.

Figure 5.9 shows how HDPV displays ownership through Reverse Reference Tracking. As the mouse is hovered over the first node in the linked list, all nodes that contain a reference to the node are highlighted, which includes the main node and detect_cycle nodes. Because a reference to the list exists in main, users can infer that detect_cycle must not destroy
void main() {
    struct node *start = NULL;
    int i;
    for (i = 0; i < 7 ;i++) {
        insert(&start);
    }
    detect_cycle(start);
}

Figure 5.8: Code Sample: Aliasing

Figure 5.9: Aliasing Effects: Snapshot of HDPV displaying the aliases to the list when hovering over the node. The list has two aliases, one in `detect_cycle` and another in `main`
the elements of the list. Once the function exits and control returns to `main`, only a single alias to the list exists and `main` can therefore destroy the elements of the list.

**Low level Programmatic Details**

Because HDPV represents the actual runtime state of the system to the user, it can represent programmatic details that are otherwise hidden from the user, such as padding inserted by a compiler.

Consider the `struct` block shown in Figure 5.10:

```c
struct temp {
    char a;
    int b;
    float c;
};
```

Figure 5.10: Code Sample: Understanding low level programmatic details

The size of the structure shown above is 12 bytes. HDPV accurately represents the memory layout of this element. Here, though the variable `a` is of size 1 in C/C++, the compiler has padded the structure with 3 bytes to align the field `b` at a multiple of 4.

Figure 5.11 shows how HDPV visualizes the structure of this block accurately. A ‘Padding’ field denotes the padding introduced by the compiler.

Figure 5.11: Snapshot of HDPV visualizing a padded memory structure
5.1.2 Demonstrating Algorithm Implementations

Finding a cycle in a linked list and reversing a linked list are two common problems encountered by students in data structure courses. This subsection illustrates how HDPV can be used by instructors to demonstrate the implementation of these algorithms.

Detecting Cycle in a linked list

Floyd’s cycle finding algorithm finds a cycle in a linked list using two pointers, a fast and a slow pointer. The implementation of this algorithm is shown in Figure 5.12. The fast pointer traverses two nodes at a time, while the slow pointer traverses one node at a time. If the two pointers collide, a cycle is detected. Students must keep track of the position of both pointers mentally in order to understand the algorithm correctly.

```c
int detect_cycle(struct node *start) {
    struct node *fast, *slow;
    slow = start;
    if (slow->next != NULL)
        fast = slow->next->next;
    else
        return 0;
    while (1) {
        if (!fast || !(fast->next) || !slow) {
            return 0;
        }
        else if (fast == slow || fast->next == slow) {
            return 1;
        }
        else {
            slow = slow->next;
            fast = fast->next->next;
        }
    }
}
```

Figure 5.12: Code Sample: Detecting cycle in a linked list
Figure 5.13 shows HDPV visualizing a linked list that has a cycle. This arrangement has been obtained by interacting with the visualization.

Figure 5.14 shows the program in the middle of cycle detection. The fast pointer moves two steps ahead of slow pointer at every iteration of the loop. Figure 5.15 shows the step before the cycle is detected.

Reversing a Linked List

The code to reverse a linked list using a recursive approach is shown in Figure 5.16. In this program, successive calls to reverse_list result in a traversal of the list until the end of the list is reached. During the traversal, the parameter elem holds a reference to individual nodes in the list. When the recursive call returns, the nodes of the list are reversed using the elem reference preserved by the recursive function.

Figure 5.17 shows the snapshot of the system after two nodes have been reversed.
Figure 5.14: Program state in the middle of the cycle detection step

Figure 5.19 shows how the same problem of reversing a linked list can be solved iteratively. The code snippet is shown in Figure 5.18.

In the iterative approach, only three pointers are required to reverse the list. The figure shows the first three nodes being reversed.

The key insights of this use case is that the space complexity of the recursive algorithm increases with the number of nodes since a new stack frame is created for each call to the recursive function. On the other hand, the space complexity is constant for the iterative approach. Therefore, the recursive approach is not a good choice to solve this problem.

Another interesting observation is that in the recursive approach, reversal of nodes proceeds from the end, whereas in the iterative approach, it proceeds from the beginning.
Figure 5.15: Linked list after cycle is detected: *fast* pointer trailing behind *slow* pointer by 1 step. This step signals a cycle which terminates the loop.

```c
struct node * reverse_list(struct node *elem) {
    struct node *ret;
    if (elem->next == NULL) {
        return elem;
    } else {
        ret = reverse_list(elem->next);
        elem->next->next = elem;
        elem->next = NULL;
        return ret;
    }
}
```

Figure 5.16: Code Sample: Reversing a linked list using a recursive approach.
struct node* reverse_list_iterative(struct node *elem) {
    struct node *next, *current = elem, *result = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = result;
        result = current;
        current = next;
    }

    return result;
}
5.1.3 Visual Debugging

Programmatic Errors

Buffer overflow errors can be identified easily if the user can visually observe the change in state of the system when the error occurs. An example of a buffer overflow error is shown in Figure 5.20. In this example, the string copy operation overwrites the value stored in the flag field when the string’s sentinel character is copied, causing a buffer overflow error.

Figure 5.21 shows the state of the object before the buffer overrun error. The object can be viewed in two representations, first as an array of characters, and second, as an object...
of type `Message`. The figure also shows the value 1 being assigned. Uninitialized memory regions are shown with a `?`.

Figure 5.22 shows the state of the object after the `strcpy()` call completes. The `flag` variable turns zero as a result of the string copy operation when the sentinel character overwrites the flag.

Figure 5.21: Buffer Overflow - Initial State: State of the object before the array overrun error. The two type representations of the object are shown side by side

Figure 5.22: Buffer Overflow - Final State: State of the object after the error

Memory Leaks

HDPV represents leaked objects that are unreachable from either a stack or global object as objects that drift off towards the right. The user can identify the source of the memory leak.
by observing the instant at which the link that holds the object is broken and the objects start drifting.

Consider the code sample shown in Figure 5.23. In this example, memory is allocated for a temp object, which is held through pointer obj. This pointer is repeatedly reassigned in the for loop without freeing the previously allocated instance, causing a memory leak.

```c
void main() {
    int i;
    for (i = 0 ; i < 10 ; i++) {
        struct temp *obj = (struct temp *)malloc(sizeof(struct temp));
    }
}
```

Figure 5.23: Code Sample: Memory Leak - 1

Figure 5.24 shows how HDPV visualizes leaked memory regions. Floating objects not held through any references signify a memory leak.

Another case of a memory leak created inadvertently by the programmer is shown in Figure 5.25. In this example, the programmer freed only the first node in a list, rather than each node. Figure 5.26 shows how HDPV visualizes this memory leak. As the first node is freed, the rest of the objects are leaked.

5.2 Performance

In this section, we discuss the performance of the Monitor and the Visualizer.
Figure 5.24: Memory Leak: Example 1: Snapshot of HDPV displaying leaked memory regions. The floating temp objects not held by any links signal leaked memory regions

```c
struct node *start = NULL;
int i;
for (i = 0; i<5 ;i++) {
    insert(&start); //insert into linked list
}

//Free the list
if (start != NULL) {
    free(start);
}
```

Figure 5.25: Code Sample: Memory Leak - 2
Figure 5.26: Memory Leak - Example 2: Memory leak that occurs if a programmer accidentally frees the first node of a linked list. The top snapshot shows the state before `free` is called. The bottom snapshot shows the state after `free` is called.

5.2.1 Performance of Monitor

The Monitor uses binary instrumentation to collect program state information. Because every memory access and function call invoked in the program is instrumented, programs experience a significant slowdown when compared to their normal execution [Gop06]. Table 5.1 shows the time taken to execute the set of use cases when the programs are run with the Monitor. Without instrumentation, these programs complete momentarily. The source code for the programs is given in Appendix 7.2.
Table 5.1: Monitor Performance: Time taken for the use cases shown in Appendix 7.2

<table>
<thead>
<tr>
<th>Use Case Name</th>
<th>Program</th>
<th>Time (in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding Recursion</td>
<td>recursion.c</td>
<td>12.59</td>
</tr>
<tr>
<td>Understanding ‘this’</td>
<td>traverse.cpp</td>
<td>25.58</td>
</tr>
<tr>
<td>Detecting a cycle in a linked list</td>
<td>cycle_detect.c</td>
<td>14.05</td>
</tr>
<tr>
<td>Reversing a linked list</td>
<td>reverse_list.c</td>
<td>13.99</td>
</tr>
<tr>
<td>Memory Leaks</td>
<td>memory_leak.c</td>
<td>13.92</td>
</tr>
<tr>
<td>Large Heap Structure</td>
<td>linked_tree.c</td>
<td>17.63</td>
</tr>
</tbody>
</table>

Table 5.2: Performance of Visualizer: XML processing and Shadow State computation

<table>
<thead>
<tr>
<th>Use Case</th>
<th># of Typedef msgs</th>
<th># of Memalloc msgs</th>
<th># of Memfree msgs</th>
<th># of Putfield msgs</th>
<th># of Showas msgs</th>
<th># of Location msgs</th>
<th>Total msg #</th>
<th>Time (in msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding Recursion</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>24</td>
<td>0</td>
<td>26</td>
<td>64</td>
<td>492</td>
</tr>
<tr>
<td>Understanding ‘this’</td>
<td>5</td>
<td>31</td>
<td>21</td>
<td>97</td>
<td>0</td>
<td>89</td>
<td>243</td>
<td>685</td>
</tr>
<tr>
<td>Detecting a cycle in a linked list</td>
<td>4</td>
<td>21</td>
<td>10</td>
<td>87</td>
<td>0</td>
<td>89</td>
<td>211</td>
<td>675</td>
</tr>
<tr>
<td>Reversing a linked list</td>
<td>4</td>
<td>17</td>
<td>11</td>
<td>65</td>
<td>0</td>
<td>67</td>
<td>164</td>
<td>689</td>
</tr>
<tr>
<td>Memory Leaks</td>
<td>2</td>
<td>11</td>
<td>1</td>
<td>24</td>
<td>0</td>
<td>2</td>
<td>40</td>
<td>484</td>
</tr>
<tr>
<td>Large Heap Structure</td>
<td>6</td>
<td>1496</td>
<td>1196</td>
<td>7308</td>
<td>0</td>
<td>6415</td>
<td>16421</td>
<td>5219</td>
</tr>
</tbody>
</table>

5.2.2 Performance of Visualizer

Table 5.2 gives the list of use cases and the corresponding time taken for reading the XML messages and updating the shadow state data structure. From the table, we find that the shadow state computation does not cause a considerable slowdown in performance. The performance bottleneck arises in the process of rendering. Currently, two actions are performed continuously by the Visualizer: (1) the force simulation that acts on the heap nodes and (2) the rendering of the visual items to the graphical context. Both actions provide potential
for optimizations.

Although the force simulation changes only the positions of floating heap nodes during its run, we currently redraw all objects. We could optimize our implementation to redraw only those nodes whose positions have changed.

In addition, the prefuse action modules that are responsible for computing the layouts and rendering the visual nodes compete for a single lock. The ensuing lock contention slows down performance considerably. By using different locks, this contention could be eliminated.

5.3 Current Limitations

1. Our Visualizer lacks a *Rewind* capability. Users can only step forward through the visualization.

2. We have not devised a specific layout strategy for concurrent programs.
Chapter 6

Related Work

This chapter compares HDPV with the existing state of the art. Price et al’s work on *A Taxonomy of Software Visualization Systems* provides a taxonomy of program visualization systems [PS93]. This taxonomy classifies the program visualization systems according to six major sections: Scope, Content, Form, Method, Interaction and Effectiveness, each of which is subdivided into characteristics. Table 6.1 gives the list of sections and the corresponding subcategories of each section.

We compare Jeliot [MMSBA04] and jGRASP [HJB04] with HDPV in the context of this taxonomy. Some sections of the taxonomy are not considered since their characteristics apply to all of the systems under consideration equally. We also added a new characteristic to the existing taxonomy under the *Method* section that discusses the data collection technique used.

**Scope**

Scope describes the general characteristics of the program visualization system, which includes the type of the system, class of programs it can visualize and its scalability and
Table 6.1: Price et. al’s classification of program visualization systems

<table>
<thead>
<tr>
<th>Section</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>System, Class, Scalability, Multiplicity, Concurrency, Benign/Disruptive</td>
</tr>
<tr>
<td>Content</td>
<td>Program/Algorithm, Code, Data, Compile/Runtime, Fidelity</td>
</tr>
<tr>
<td>Form</td>
<td>Medium, Graphical Elements, Color, Animation, Multiple Views, Other Modalities</td>
</tr>
<tr>
<td>Method</td>
<td>Specification Style, Batch/Live, Code Familiarity, Program Invasion, Language Customization, Same Language</td>
</tr>
<tr>
<td>Interaction</td>
<td>Navigation, Elision, Temporal Control Mapping</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>Appropriateness and Clarity, Experimental Evaluation, Production Use</td>
</tr>
</tbody>
</table>

Concurrency properties.

**Class of Program:** Different systems are targeted for different classes of programs. Jeliot visualizes Java programs. Though jGRASP’s Integrated Development Environment (IDE) supports programs written in C, C++, Java, and Ada, dynamic object viewers are supported only for Java. jGRASP supports the generation of Control Structure Diagrams (CSD) for Java, C, and C++.

By contrast, HDPV is not tied to a particular programming language. However, language-specific program monitors must be written to extract the program information in a format that conforms to HDPV’s intermediate language specification.

**Concurrency:** This characteristic specifies if the visualization system can visualize the execution of concurrent programs. In jGRASP, the thread section lists all active threads running in the program. Color codes describe the state of the thread. A red color code specifies a stopped thread and a green color code represents a running thread.
HDPV currently does not have a specific layout strategy for visualizing concurrent programs. However, visualization of concurrent programs could be implemented in HDPV with moderate effort by representing the stack frames of the individual threads through a suitable layout strategy.

Method

Method denotes the way in which the visualization is produced.

Data Collection Technique: jGRASP uses the Java debugging interface to collect program information, which involves setting breakpoints and inspecting the program state when the breakpoints are reached. Jeliot uses a source interpreter, Dynamic Java [Hil]. The use of a source interpreter allows Jeliot to visualize the evaluation of expressions.

HDPV uses binary instrumentation using Pin [LCM+05] for C/C++ programs and using the ASM framework [EBC02] for Java programs.

Dependency on Source Code: Jeliot needs access to the source code of the program due to its use of a source interpreter to collect program information, whereas jGRASP does not have this restriction.

The C/C++ Program Monitor of HDPV can operate directly on the native C/C++ binary. The Java Monitor can operate on the Java class files. Therefore, access to source code is not required.

Content

The Content specifies the characteristics of the visualization produced by the system.
**Type of Visualization:** Software Visualization Systems can produce different types of visualization such as program visualization, algorithm visualization, or source code visualization. Systems such as Jeliot produce visualizations of program state. Jeliot also animates the source code that is currently being executed. jGRASP produces Control Structure Diagrams (CSD) \[IBHT96\], which visualize the control structure of the source code. jGRASP provides this feature through an Integrated Development Environment (IDE). Apart from CSD, jGRASP produces visualizations of data structures through animated object viewers. HDPV is designed for the visualization of program state. In addition, HDPV can highlight the line of code that is currently under execution.

**Fidelity and Completeness:** This characteristic defines if the visual representation presented by the tool provides a faithful image of the underlying runtime system on which it operates. From the existing literature, we find that none of the current tools represents a completely faithful picture of the underlying runtime system. For instance, although Jeliot represents the stack frame of a function as part of the *Method Area* section of the visual space, the order in which the variables are displayed has no relationship to the actual layout of the variables in the stack frame of the runtime system. jGRASP also lacks fidelity. This lack of fidelity occurs because the actual runtime state of the program is not exposed through the data collection techniques used by these systems.

HDPV can provide Fidelity because it uses binary instrumentation to extract the actual runtime state.

**Method**

The *Method* specifies the way in which users can interact with the program visualization systems.
Specification Style: Most of the current program visualization systems operate without any intervention from the user. Past algorithm animation systems such as Balsa [Bro88] and Zeus [Bro91] required the user to annotate the program source with animation routines. Because this approach requires the user to learn a new language for annotating the program, it is discouraged in modern visualization systems.

HDPV does not require any user intervention.

User Customization: Jeliot does not allow users to customize the visualization. On the other hand, jGRASP provides visualization customization through animated object viewers. In addition to a number of built-in viewers, users can also create and use custom animation viewers.

HDPV allows users to customize the layout of the visual objects through the fix and drag controls. In addition, the Multi-Type Representation Control allows users to visualize a memory region in multiple type representations as described in subsection 3.4.4.

Interaction

This section specifies the amount of interactive capabilities provided by the visualization system to its users.

Navigation: The existing state of art provides limited capabilities for navigating through a large data set. System such as Jeliot uses scrollbars for navigation. jGRASP provide navigation through a combination of Zoom and scrollbars to inspect different parts of the visualization. Systems using Scrollbars suffer from a loss of context when users navigate to different parts of the visualization space.

HDPV supports navigating large data sets through a Zoom + Pan control as well as the
Elision: Elision is based on the concept of Details on Demand through which information is elided or detailed information is suppressed to prevent the display being cluttered with excessive information. Jeliot does not provide this feature, whereas jGRASP uses abbreviations to support elision.

HDPV provides elision through a combination of abbreviations and tooltips. Excessively long identifiers are abbreviated. The abbreviated identifiers are displayed in its entirety through the use of tooltips.

Effectiveness

Experimental evaluation of jGRASP found a significant improvement in student performance [JJHCH06]. Jeliot’s evaluation showed that the tool can be used in introductory programming course and students found the visualizations useful [LTM00].

HDPV has not yet been subjected to a user study.

Summary

Table 6.2 summarizes the comparison of Jeliot, jGRASP, and HDPV according to the characteristics discussed in this section.
Table 6.2: Comparison of existing program visualization systems with HDPV

<table>
<thead>
<tr>
<th>Section</th>
<th>Characteristic</th>
<th>Jeliot</th>
<th>JGrasp</th>
<th>HDPV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Class of Program</td>
<td>Java</td>
<td>Java</td>
<td>Language Independent</td>
</tr>
<tr>
<td></td>
<td>Concurrency</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Source</td>
<td>Data Collection Technique</td>
<td>Source Interpreter</td>
<td>Debugging Interface</td>
<td>Binary Instrumentation</td>
</tr>
<tr>
<td></td>
<td>Dependency on Source Code</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Content</td>
<td>Type of Visualization</td>
<td>Code visualization, Data</td>
<td>Code Visualization including</td>
<td>Code Visualization, Data</td>
</tr>
<tr>
<td></td>
<td>Fidelity and Completeness</td>
<td>No</td>
<td>Control Structure Diagrams,</td>
<td>Visualization</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data Visualization</td>
<td>Yes</td>
</tr>
<tr>
<td>Method</td>
<td>Specification Style</td>
<td>Automatic (No User Intervention)</td>
<td>Automatic (No User intervention)</td>
<td>Automatic (No User intervention)</td>
</tr>
<tr>
<td></td>
<td>User Customization</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Interaction</td>
<td>Navigation</td>
<td>Scrollbars</td>
<td>Zoom+Scrollbars</td>
<td>Zoom+Pan</td>
</tr>
<tr>
<td></td>
<td>Elision</td>
<td>No</td>
<td>Minimal</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Chapter 7

Conclusions and Future Work

7.1 Conclusions

We have developed HDPV, a program state visualization system that visualizes programs written in multiple languages including C, C++, and Java. HDPV’s interactive and layout capabilities let users involve themselves cognitively in the visualization. Through a set of use cases, we have shown that HDPV can be employed in teaching as well as in software engineering applications.

Steven Reiss in his paper, The Paradox of Software Visualization [Rei05] states several reasons why software visualization has not attained widespread success. HDPV addresses many of the reasons Reiss identifies, such as lack of scalability or the need to abstract away irrelevant details from the visualization space. Reiss also points out that software visualizations should not require programmer intervention, a property provided by HDPV.

Several conclusions can be drawn from our work.

1. A review of the existing literature on program visualization systems shows that current
systems suffer from a number of important limitations such as lack of interactivity, lack of support for custom layouts, and dependence on a specific programming language.

2. The data collection technique employed strongly influences the characteristic of the visualization system. The use of binary instrumentation allows the extraction of program state in-vivo.

3. The use of an intermediate representation allows the construction of language independent visualization systems.

4. Based on the example use cases, we conclude that meaningful visualizations of algorithms and data structures can be built automatically or with minimal user interaction.

7.2 Future Work

There are several directions for future work.

1. HDPV has not been subjected to a user study. A user study is required to examine the effectiveness of our tool. It will also help us to identify enhancements that could be applied to it.

2. HDPV is aimed at handling large program structures whose visualization could benefit from Giga pixel displays. We plan to deploy HDPV on a Giga pixel display.

3. HDPV could be extended to visualize concurrent programs.

4. HDPV currently lacks the capability to specify different levels of granularity when collecting the program information. In the future, we plan to provide this capability.

5. Currently, HDPV does not have a *rewind* capability. A *Rewind* capability would help users step back in the visualization.
6. A *Record* capability could be added to record the visualization. This capability would facilitate the creation of animations.

7. We planned to deploy HDPV as a supplementary learning resource to students taking Data Structures and Object Oriented Design course at Virginia Tech.
Bibliography


[EGV94] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.


M. Lattu, J. Tarhio, and V. Meisalo. How a visualization tool can be used - evaluating a tool in a research development project, 2000.


Appendix A

Source Code for Use Cases

A.1 recursion.c

```c
int factorial(int n) {
    int result;
    if (n == 1)
        result = 1;
    else
        result = n * factorial(n - 1);
    return result;
}

int main() {
    int result = factorial(7);
}
```
A.2 memory_leak.c

```c
#include <stdlib.h>

struct temp {
    int a;
    float b;
};

void main() {
    int i;
    for (i = 0; i < 10; i++) {
        struct temp *obj = (struct temp *)malloc(sizeof(struct temp));
    }
}
```
#include <stdio.h>

struct node {
    struct node *next;
};

void insert(struct node **start) {
    if (start == NULL) {
        *start = (struct node *)malloc(sizeof(struct node));
        (*start)->next = NULL;
    } else {
        struct node *newNode = (struct node *)malloc(sizeof(struct node));
        if (!newNode) return;
        newNode->next = *start;
        *start = newNode;
    }
}

int detect_cycle(struct node *start) {
    struct node *fast, *slow;
    slow = start;
    if (slow->next != NULL) {
        fast = slow->next->next;
    } else {
        return 0;
    }
    while (1) {
        if (!fast || !(fast->next) || !slow) {
            return 0;
        } else if (fast == slow || fast->next == slow) {
            return 1;
        } else {
            slow = slow->next;
            fast = fast->next->next;
        }
    }
}
void main() {
    struct node *start = NULL;
    int i;
    for (i = 0; i < 7 ;i++) {
        insert(&start);
    }
    detect_cycle(start);
}
#include <stdio.h>

class Node {

    Node *left;
    Node *right;

public:
    Node(Node *left, Node *right) {
        this->left = left;
        this->right = right;
    }

    Node() {
        this->left = NULL;
        this->right = NULL;
    }

    void traverse() {
        if (this->left)
            this->left->traverse();
        if (this->right)
            this->right->traverse();
    }
};

int main() {

    Node *root =
        new Node(
            new Node(
                new Node(),
                NULL),
            new Node(
                new Node()
                new Node()));

    root->traverse();
    return 0;
}
#include <stdio.h>

class Cell {
    Cell *next;
    public:
    Cell() {
        next = NULL;
    }
    Cell(Cell *next) {
        this->next = next;
    }
};

int main() {
    Cell *list =
        new Cell(
            new Cell(
                new Cell(
                    new Cell(
                        new Cell())));
    return 0;
}
A.6 reverse_list.cpp

```c
#include <stdio.h>

struct node {
    struct node *next;
};

void insert(struct node **start) {
    if (start == NULL) {
        *start = (struct node *)malloc(sizeof(struct node));
        (*start)->next = NULL;
    } else {
        struct node *newNode = (struct node *)malloc(sizeof(struct node));
        newNode->next = *start;
        *start = newNode;
    }
}

struct node * reverse_list_recursive(struct node *elem) {
    struct node *ret;
    if (elem->next == NULL){
        return elem;
    } else {
        ret = reverse_list(elem->next);
        elem->next->next = elem;
        elem->next = NULL;
        return ret;
    }
}

struct node* reverse_list_iterative(struct node *elem) {
    struct node *next, *current = elem, *result = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = result;
        result = current;
        current = next;
    }
    return result;
}

void main() {
    struct node *start = NULL;
```
int i;
for (i = 0; i < 5; i++) {
    insert(&start);
}
reverse_list_recursive(start);
reverse_list_iterative(start);
A.7  linked_tree.cpp

```c
#include <stdio.h>
#include <stdlib.h>

/* Node of a tree */
struct node {
    int data;
    struct node* left;
    struct node* right;
};

/* Node of a list */
struct link_node {
    struct node *tree;
    struct link_node *next;
};

struct node* newNode(int data) {
    struct node* newnode = (struct node *)malloc(sizeof(struct node));
    newnode->data = data;
    newnode->left = newnode->right = NULL;
    return newnode;
}

struct node* insert(struct node* node, int data) {
    if (node == NULL)
        return newNode(data);
    else {
        if (data <= node->data)
            node->left = insert(node->left, data);
        else
            node->right = insert(node->right, data);
        return node;
    }
}

struct node* create_tree() {
    int i;
    struct node *root = (struct node *)malloc(sizeof(struct node));
    root->left = node->right = NULL;
    root->data = rand();
    for (i=0 ; i < 10 ; i++)
        insert(root, rand()); //Insert Node into the tree
    return root;
}

void main(){
```
```c
int idx;
struct link_node *start = NULL, *temp = NULL;
for (idx = 0; idx < 25; idx++){
    temp = (struct link_node *)malloc(sizeof(struct link_node));
    temp->tree = create_tree();
    if (start == NULL){ //Start node
        start = temp; //Start node
        start->next = NULL;
    } else { //Insert the node into the list
        temp->next = start;
        start = temp;
    }
}
```