SOFTVIZ... A STEP FORWARD

by

Mahim Singh

A Thesis

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

April 2004

APPROVED:

____

Professor George T. Heineman, Thesis Advisor

____

Professor Matthew O. Ward, Thesis Co-advisor

____

Professor Gary F. Pollice, Thesis Reader

____

Professor Michael A. Gennert, Head of Department

Abstract

Complex software systems are difficult to understand and very hard to debug. Programmers trying to understand or debug these systems must read through source code which may span over thousands of files. Software Visualization tries to ease this burden by using graphics and animation to convey important information about the program to the user, which may be used either for understanding the behavior of the program or for detecting any defects within the code. SoftViz is one such software visualization system, developed by Ben Kurtz under the guidance of Prof. George T. Heineman at WPI. We carry forward the work initiated with SoftViz. Our preliminary study showed various avenues for making the system more effective and user-friendly. Specifically I completed the unfinished work, made optimizations, implemented new functionality and added new visualization plug-ins, all aimed at making the system a more versatile and user-friendly debugging tool. We built a solid core functionality that would be able to support various functionalities and created new plug-ins that would make understanding and bug-detection easier. Further we integrated SoftViz with the Eclipse development environment, making the system easily accessible and potentially widely used. We created an error classification framework relating the common error classes and the visualizations that could be used to detect them. We believe this will be helpful in both selecting the right visualization options as well as constructing new plug-ins.

Acknowledgements

I take this opportunity to thank all the people who helped me in my work. A special thanks to SERG (Software Engineering Reasearch Group) members for their suggestions and recommendations to make this work successful.

I express my greatest gratitude to my advisors, Prof. George T. Heineman and Prof. Matthew O. Ward, for their guidance and invaluable contributions to this work. I consider myself very fortunate that I have two such excellent professors as my advisors.

I thank Prof. Gary F. Pollice for being the reader of this thesis and giving me much valuable feedback. He has been a great support all along the way and provided much needed assistance with Eclipse.

This work would have been impossible without the loving care of my family and friends. Special thanks to Anil and Nishant for providing help with Java and LaTeX.

Without all of you, I couldn't have done it. Thank You !

Contents

List of Figures

Chapter 1

Introduction

The complexity of software systems has increased steadily, and so has the need for alternatives to conventional text-based debugging techniques. Many challenges posed by the dynamic behavior of object-oriented code cannot be met by simply reading code. Software Visualization may hold the key to these problems [29, 40, 13, 43, 42].

SoftViz [31] is one such software visualization system developed at WPI. The original version had a working infrastructure in place; however an evaluation of the system brought forward various limitations, the main ones being the usage of the heavy weight SIENA [48] architecture for event delivery, the code-intrusive paradigm for event extraction and the stand-alone design of the system. Many features (such as the stream storage and retrieval, stored stream manipulation and stream control panel) were only partially implemented. Further avenues existed to incorporate new functionality, such as the event stream buffering and inter-plug-in communication. This thesis carries forward the work started with the development of SoftViz, overcoming the limitations of the original version, and adds new functionality to make SoftViz a better debugging platform.

1.1 SoftViz - The original system

SoftViz was developed as a tool for program understanding as well as visual debugging [31]. The aim was two-fold, to assist new developers trying to familiarize themselves with a complex system and support experienced developers working on debugging and optimization. SoftViz allows users to analyze the runtime behavior as well as visualize the static structure of their software systems. SoftViz extracted run-time events of interest using a probe-based approach and published events using the SIENA [48] wide-area event distribution bus.

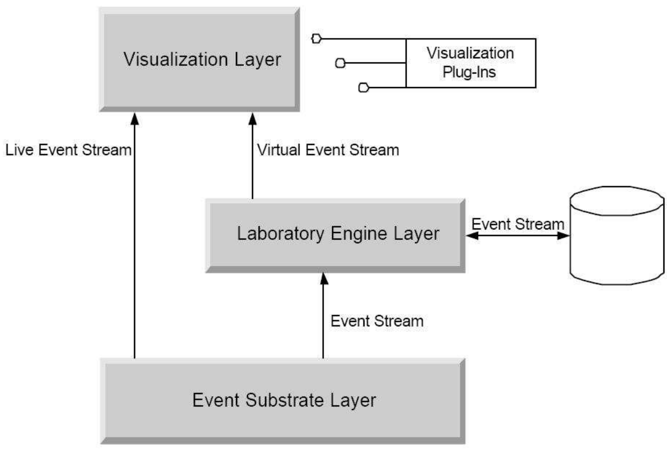


Figure 1.1: Three-layered architecture of SoftViz.

SoftViz had a three-layered architecture, as seen in Figure 1.1. The Event Substrate Layer extracted events from the software system under observation. The Laboratory Engine Layer filtered and aggregated event streams and could store events from live streams for playback at a later time. The Visualization Layer combined live or virtual event streams with the structural information about the instrumented program to produce the final visualization. The system was designed to support visualization plug-ins, independent program modules that accept event stream data from SoftViz and produce their own visualizations. New plug-ins could be designed and customized to satisfy the specific user needs, which may vary depending upon whether the user needs to learn about the system or debug it. The system could thus evolve continuously and could be tailored to meet divergent requirements.

The original system had only two plug-ins, *Sunburst* and *Timeline*. The Sunburst [50] visualization shown in Figure 1.2 represents a class of visualizations capable of representing a hierarchical data set in an intuitive and space-efficient manner. Sunburst conveys the layered, hierarchical aspects of Java source code through spatial representations. Sunburst used animation to show information as elements executed (by flashing regions within the visualization) and allowed user interaction such as selecting individual elements. The goal of Sunburst is to provide the user with a way of understanding both the static, hierarchical structure of a software system and the dynamic, control-flow of the system as it runs. Once the static hierarchy of the system is represented with Sunburst, the flow of control between these elements is shown by raising the brightness and saturation of the element that is currently executing. The varying of brightness and saturation happens gradually across several seconds, creating a pulse effect. This allows the viewer of the visualization to follow the flow of the program.

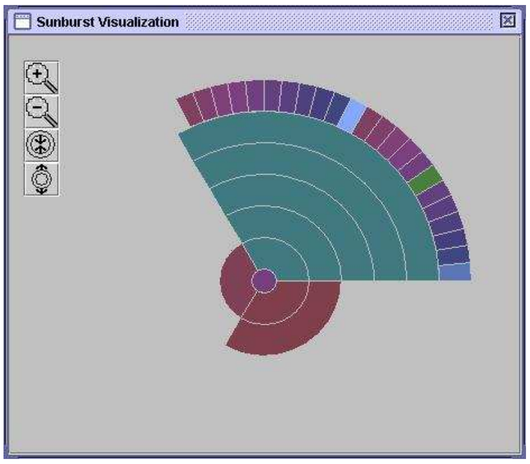


Figure 1.2: The Sunburst Visualization.

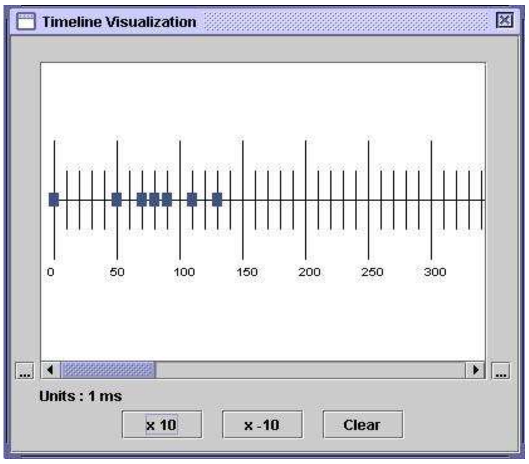


Figure 1.3: The Timeline Visualization

The Timeline visualization, as seen in Figure 1.3, shows dynamic events on a timeline, colored according to the associated structural element. It conveys the relationship of events along a temporal axis and contains additional information about the source of each event in the source code. As each event occurs, it appears on the timeline as a colored bar. The color of the bar corresponds to the structural element from which the event originated. This visualization module provides no information about the static structure of the system in itself, but provides a dynamic insight into temporal relationships of events, including the time between events and their ordering.

1.2 Evaluation of SoftViz

While SoftViz met its stated objectives [31], three limitations have been found:

1. Assumptions made by SoftViz,
2. Only a subset of SoftViz was implemented, and
3. The desired extension to SoftViz required major reimplementation and some redesign.

1.2.1 Assumptions made by SoftViz

SoftViz was originally code-intrusive, meaning it required changes to the source-code of the application under observation. It was required to instrument the application's source code with probes (small pieces of code that generated call-backs as the code executed). Although this could be done automatically using AIDE [21], instrumentation was a time-consuming process. Further, the cost of instrumenting the code and building it each time the application's source code was changed made it infeasible to use the system while the application was under development.

SoftViz was a stand-alone system, which limited its usability as a debugging tool. The systems used for debugging are generally closely related with the development environment so that the user may switch between coding and debugging frequently, which is how the development naturally takes place.

The SIENA event distribution bus used for event delivery was both slow and inefficient, and induced a lot of overhead. SIENA was capable of processing only a few hundred events per second. This caused a major bottleneck right at the source and slowed down the whole system.

There were only two plug-ins in the repertoire, and they did not prove especially helpful in detecting any kind of defects. Sunburst was a good static visualization plug-in aimed at visualizing hierarchical data sets, although it was not effective for visualizing dynamic information. It was configured to present dynamic information at class level; a view that was too high level, incapable of helping in any real analysis. Timeline visualized class level information in the context of time, again presenting information that was too abstract.

1.2.2 Only a subset of SoftViz was implemented

Functionality related to stored event streams (such as the event storage and retrieval) were only partially implemented.

There was no provision for buffering the events within the system. This implied that the plug-ins either had no access to the past events (hence limiting their scope immensely), or they had to buffer events within themselves. Buffering of events within the plug-in may look like a good option, since the buffers can then be customized to the needs of the specific plug-in and optimized for speed; however storing tens of thousands of events within each plug-in, where multiple plug-ins may be running simultaneously, poses a scalability issue.

Neither Sunburst nor Timeline were fully developed. Various features, especially the ones related to abstraction such as scaling, zooming and panning, were absent. Many functionalities were only partially implemented and there was a lot of room for improvement, especially in Timeline.

1.2.3 Extensions required reimplementation and redesign

There were no guidelines for a developer to create plug-ins dedicated to solving particular problems. What was required was a formal set of specifications or recommendations for developing plug-ins that specialize in detecting particular classes of defects.

Many GUIs needed to be updated, which required a lot of redesign and implementation. The main splash screen (SoftViz main window) needed to include the VCR-like control panel for user controlled stream manipulation. The Timeline GUI needed to be redesigned to provide access to the new features.

A new design was required to incorporate the inter-plug-in communication channel. Many core parts of the system had to be reimplemented to accommodate the new functionality. Further, all the plug-ins had to be redesigned, and their GUIs needed to be updated.

1.3 Motivation

It has been pointed out by French [19] that most advancement in software engineering has focussed on "writing of programs" resulting in better and efficient assemblers, high level language compilers and compiler-compilers; however, understanding and/or debugging of programs have been largely ignored. French claims that "debugging interactive programs is the most difficult of all" and "core dumps (post-mortems), traces, print statements etc. are simply not sufficient" to understand/debug such systems. It is also claimed that "the time required to understand or debug a program is at least equal to the writing time and in interactive programs can be many times longer”.

The basic need of engineers involved in debugging and novices trying to understand a system remains to “understand the system”. However typical software systems of this day consist of millions of lines of code spread over hundreds or thousands of files. Such huge size coupled with complexity prohibit conventional techniques of system understanding such as reading code and single stepping execution. Simplification of the system can greatly assist understanding and tools are required that can reproduce the system behavior at a higher level of abstraction.

This section is divided into three subsections. The first introduces the theory behind debugging and defines certain terms used in further discussions. The next outlines the motivation behind software visualization being used for debugging, and the last subsection brings forward the motivation for using software visualization for program understanding.

1.3.1 Defects, errors and faults

The theory behind debugging adopted by this work is the fault/error/defect model as specified in [11]. The model defines -

Fault: Observed deviation between software specification and its actual behavior. A fault is triggered by one or more errors.

Error: Manifestation of a defect at run-time.

Defect: A design or coding mistake that may cause abnormal software behavior.

According to this model, the goal of any debugging process is to discover a defect in the source. The observation of a fault is what informs the user that a defect may exist. An error is the behavior that the software exhibits due to the presence of the defect.

1.3.2 Debugging

Conventional techniques such as setting breakpoints, line-by-line debugging, observing the call stack and code inspection do not scale well with the huge software systems of this day. Where they provide much detailed information about the program, these techniques are just too low level for locating faults in the code. Command line debugging tools such as GNU's gdb [20], Berkeley UNIX symbolic debugger dbx, WDB, JDB, XDB, the Perl debugger, and the Python debugger allow the user to single step through the code and inspect the program state as each line is executed. Almost all development environments, such as Visual Age, Visual Cafe, Microsoft's Visual J++, JBuilder and the most recent development platform Eclipse from IBM, provide integrated debugging facilities. They provide convenient access to the debugging information, although they too rely on the conventional techniques such as the ones described above. Therefore the users are still faced with similar challenges, although they have some relief as they interact with a more convenient graphical interface. Many believe that Software Visualization, using techniques such as abstraction and elision, may help users detect the faults quickly, easily and effortlessly [29, 40, 13, 43, 42]. Dynamic visualizations may rely on known manifestations of specific error types and may be custom-made to detect them. The argument in favor of this technique is that the onus is no more on the user but on the visualization module, and this shifts most of the mundane tasks, traditionally a responsibility of the user, to the module itself.

As an analogy, consider detecting faults to be like finding a needle in a haystack. The conventional techniques recommend inspecting each strain of hay (piece of code) in an effort to locate the needle (defect). This would eventually result in finding the fault, however imagine the drudgery of the whole process, not to mention all the time and resources wasted upon this process. Software Visualization, on the other hand tries to differentiate the fault from good piece of code. In the analogy, we may imagine a piece of magnet being used to find the needle, which is basically relying on the specific property of the defect (needle) that separates it from the rest of the code (hay), that being its attraction to the magnet. We may develop specialized visualizations that separate out particular categories of faults, depending upon specific properties of the fault.

1.3.3 Understanding

System understanding has long been a standard task in the industry, whether it be for a new recruit in the company or a group of engineers assigned the job of maintaining a system that they did not develop themselves. The first step in debugging is system understanding; it is not possible to locate most defects without thoroughly understanding what the system is doing and what it is supposed to do. However not much work has been done to aid system understanding by itself; most of the work has been a result of the research on software debugging. The arguments presented in favor of debugging apply to system understanding as well.

Code understanding includes understanding the structure of the code as well as knowing what code executes (for what actions/inputs/conditions). SoftViz helps code understanding by providing a gross behavioral structure of the execution. Ordering is important, and many profiling tools do not provide any information about the order of events captured at run-time. SoftViz preserves ordering, and displays the same using its plug-in Timeline. However the current visualization options present a view of the system that is too high level, which does not particularly benefit understanding of the system.

1.4 Problem being solved

We believe that using conventional debugging techniques such as line-by-line debugging, breakpoints and frequent inspection of variable values is tedious. The thesis claims that

SoftViz improves debugging ability by relying on visualizations to present an image that lets the user realize that a fault has occurred. This claim is supported by Mukherjea and Stasko [36] who believe that “pictures help make an abstract concept more concrete”.

Conventional line-by-line debugging techniques assume that the user knows what is supposed to happen and knows the location in the code where the fault can be observed to some extent. This is supported by the fact that the user sets breakpoints or observes the line-by-line execution at some particular location out of all the available source code. If the location where the fault occurred is not exactly known, the user either sets the breakpoint early enough and then single-steps through the code, or sets the breakpoint late and backtracks from there. Further, there is an implicit assumption that the user can spot the error behind the fault by:

1. Detecting which line executes (for line-by-line debugging),
2. Looking at the variable values (for variable inspection techniques), and
3. Inspecting the call stack (for techniques that present the call-stack).

The most important aspect of debugging is seldom addressed, namely the mental model developed by the user. It is this model that guides the user towards the location of the error. As the debugging session proceeds, the user builds up a program state/behavior in his/her head and this mental model of the behavior is compared with a model of “what should happen”. The goal of debugging is to detect where the correspondence between the two models is lost, and to “triangulate” backwards from the fault to an error (or set of errors) to ultimately locate the defect in the code. The overall success of debugging is dependent on the accuracy of the user’s mental model.

1.5 Contributions

This thesis contributes to the field of Software Visualization and software debugging. It presents an enhanced SoftViz system that provides more flexibility, more features and better debugging techniques. The existing plug-ins, Sunburst and Timeline, have been enhanced. The plug-in Timeline has new abstraction techniques and provides flexible controls. The system's repertoire of visualization plug-ins has been augmented with two new plug-ins. We have successfully integrated SoftViz with a well known development environment (Eclipse) that has allowed it to be easily accessible and more readily usable than ever before.

This thesis brings forward an error classification framework that categorizes the common programming errors and proposes visualization options that exist or may be developed in the future to help detect/debug these errors. This would assist in the development of specialized visualization plug-ins.

1.6 Thesis Outline

This thesis is divided into six chapters. Chapter 2 describes the related research in the fields of software visualization, program understanding and debugging and lists other software visualization packages that have been developed. Chapter 3 outlines the enhancements made to SoftViz. Integration of SoftViz with the Eclipse IDE is discussed in Chapter 4. Chapter 5 introduces the Error Classification Framework that relates error classes with effective visualization options. Two new plug-ins contributed by this thesis to SoftViz are described in Chapter 6. Finally, the conclusions of the thesis and open issues for future work are described in Chapter 7.

Chapter 2

Related Work

Software visualization is a technology aimed at presenting computer programs, processes and algorithms by using computer graphics and animation [40]. In [43], Price, Baecker and Small define softare visualization as “the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software.” It has been suggested in [29, 40, 13, 43, 42] that visualization aides better understanding of the software. Software visualization can not only be used to understand how algorithms work, but also to aid software development by helping developers understand their code better.

Extensive anthologies about software visualization have been published, the most prominent ones being [13] and [53] published in 1996 and 1998 respectively. [53] contains revised versions of some seminal papers on classical algorithm animation systems as well as educational and design topics. Various aspects of software systems, both static and dynamic, have been visualized in the past. Static visualization systems focus on visualization of system structure, class models and package hierarchy. According to [12], static visualization systems are based on a static analysis of the source code and its associated information and provide a more generic high-level view of the system. Run-time behavior of systems has been captured using dynamic visualization tools. As per [12], dynamic visualization is based on information from the analysis of recorded or monitored program execution; and depending on the available run-time information, dynamic visualizations can provide a more detailed and insightful view of the system with respect to a particular program execution.

Software visualization can be categorized mainly into algorithm animation, program visualization based on content, and program understanding and debugging based on application. Price, Baecker and Small [43] distinguish between algorithm animation and program animation. Algorithm animation refers to a dynamic visualization of the higher level descriptions of software (algorithms) that are later implemented in code. On the other hand, program visualization refers to the use of dynamic visualization techniques to enhance human understanding of the actual implementation of programs or data structures. Various aspects of software systems have been visualized in the past. Static visualization systems focus on visualization of system structure, class models and package hierarchy. Run-time behavior of systems has been captured using dynamic visualization tools. Recently performance visualization tools have been developed to monitor system resources and performance metrics.

2.1 Algorithm Animation

BALSA [10, 9] and its latest evolution Zeus [7, 8], Polka [40] and PARADE [52] are all event-driven algorithm animation systems, the latter two also being concurrent program animation systems. BALSA (Brown ALgorithm Simulator and Animator) was the first widely known algorithm animation system, developed by Marc Brown and Robert Sedgewick at Brown University. BALSA was an interactive algorithm animation system that supported multiple simultaneous views of an algorithm's data structures and could display multiple algorithms executing simultaneously. Its evolution motivated other researchers to participate in the development of other algorithm animation systems. TANGO (Transition-based Animation GeneratiOn) [51] was developed by John Stasko at Brown University. TANGO introduced the path-transition paradigm for animation design and a new conceptual framework for algorithm animation systems which was adopted by many later systems as their fundamental architecture.

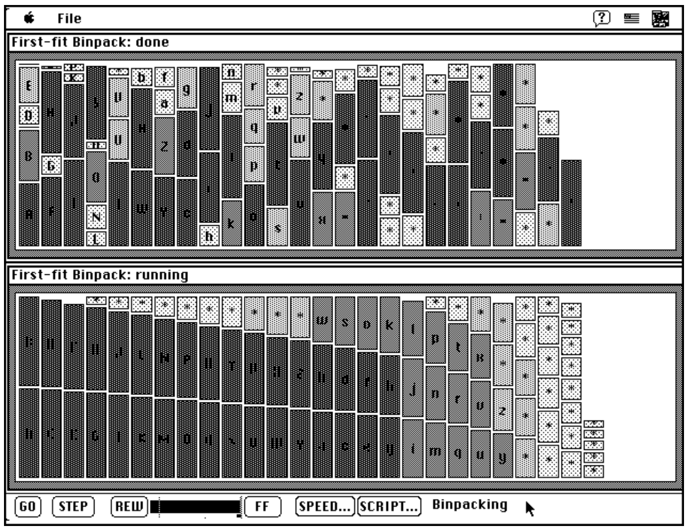


Figure 2.1: A still from Balsa-II showing a race between two bin packing programs (first-fit and first-fit decreasing) [1]

BALSA-II [6], a descendent of BALSA, was a domain-independent algorithm animation system that provided features such as step and break points, multiple views and a provision for scripting. Unlike BALSA which displayed information in black and white, BALSA-II used color and allowed rudimentary sounds to be associated with the events [1]. Figure 2.1 shows a still from an algorithm race in BALSA-II. XTANGO [49], an X window version of TANGO, utilized the path-transition paradigm to achieve smooth animation. POLKA [40] was a descendent of XTANGO, designed to build concurrent animations for parallel programs. Figure 2.2 displays a screenshot of the POLKA animation system, showing a variety of performance views [41]. Polka, originally an object-oriented 2-D algorithm animation system, was extended to be a 3-D system, POLKA 3-D. POLKA 3-D provided 3-D views and 3-D primitives such as cones, spheres, cubes and more. Figure 2.3 displays the 3D view of a quicksort algorithm generated by POLKA 3-D. The blue dots to the right represent the values being sorted, and the multicolored planes to the left provide a history of the exchanges in the program [39].

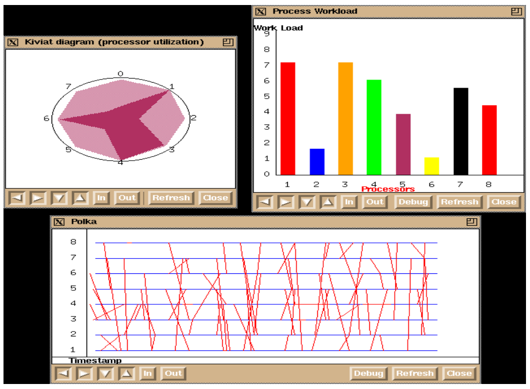


Figure 2.2: An animation showing a variety of performance views built with POLKA: a Kiviat diagram, a workload bar chart, and a Feynman diagram [41]

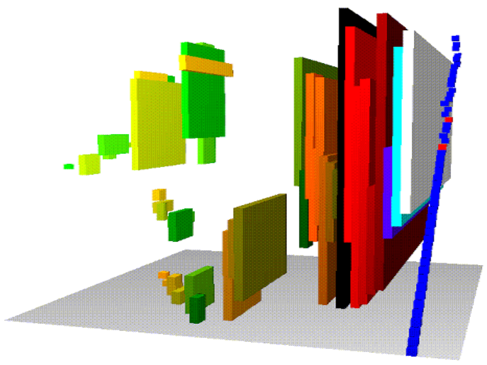


Figure 2.3: POLKA : 3-D view of a quicksort algorithm [39]

Zeus [7] was developed at Brown University, concurrently with BALSA and BALSA-II. It was one of the first major interactive software visualization systems. It supported multiple synchronized views and allowed editing of the views and changing a data item's representation. Zeus was implemented in a multi-threaded and multi-processor environment, so it could animate parallel programs. Like POLKA, Zeus was extended into a 3-D system. Figure 2.4 displays a screenshot of Zeus comparing six different online algorithms with the optimal offline algorithm. The gray-blue bars on the left show the waiting time; the green and red bars show the spinning/blocking time used by each algorithm. The vertical red lines show the current thresholds at which each algorithm stops spinning and decides to block. The performance of each method is made visible graphically by the progress downward in each column. The view at the bottom of the screen shows the performance ratios numerically and graphically [60]. Figure 2.5 shows Zeus visualizing a hashing scheme that uses multiple hash tables to store an online dictionary. Elements in the “Keys” view are color-coded to indicate the tables to which they belong in the “Hash Tables” view. The color of the band surrounding each hash table corresponds to the hash function currently in use for that table. The two statistics views on the right show the number of percolations out of the table and the table load for each hash table. The “Sound Effects” view causes a table-specific tone to be sounded each time an element is inserted into a hash table [61].

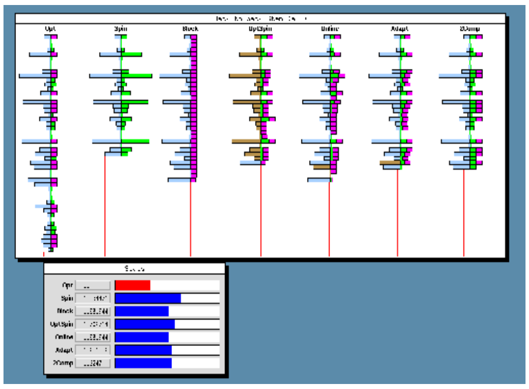


Figure 2.4: Zeus compares six different online algorithms with an optimal offline algorithm [60]

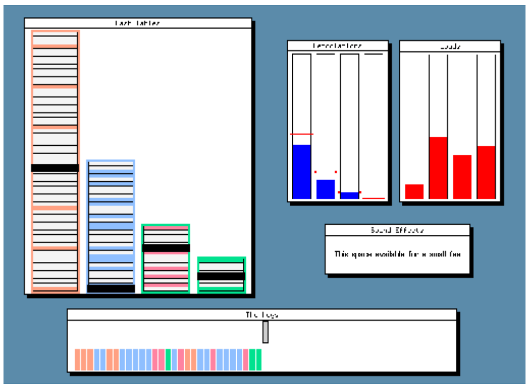


Figure 2.5: Zeus screenshot illustrating a hashing scheme that uses multiple hash tables to store an online dictionary [61]

Post-mortem systems such as ANIM [4] (shown in Figure 2.6) generate visualizations only after the execution ends. ANIM was a simple algorithm animation program that could create simple, but useful animations. ANIM could be used to visualize programs in any language, since it worked by inserting ANIM statements within the algorithm code, in the way of print statements. These print statements were used to generate an output script file that was then read using the movie tool or stills tool, to provide a dynamic animation or snapshots of the algorithm at different points in time.

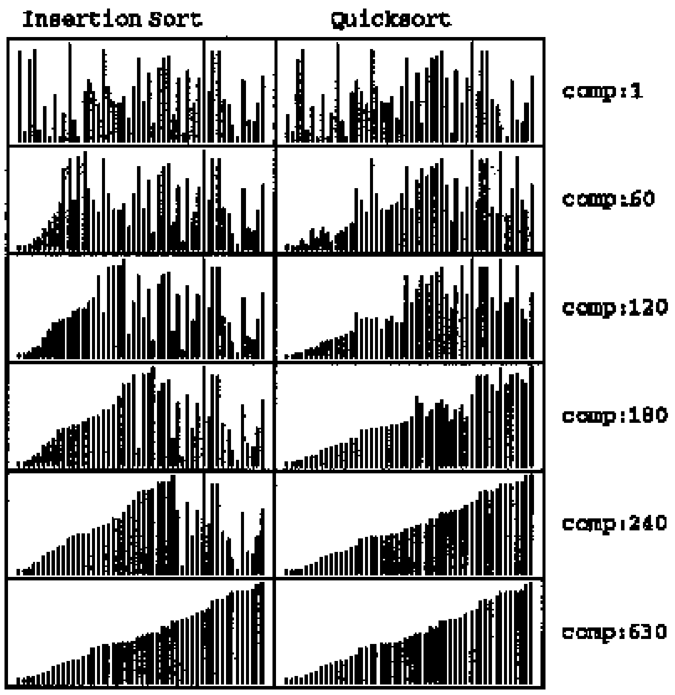


Figure 2.6: ANIM : A post-mortem visualization of insertion-sort and quicksort algorithms.

2.2 Program Visualization

Program Visualization systems are generally real-time systems that produce visualizations as the target program executes. Examples of real-time systems include PV (Program Visualization) [30], Jinsight [47] and the recent Java Visualizer [46]. PV was a prototype program visualization system developed at IBM Research. It provided continuous visual displays of the behavior of a program and its underlying system. PV displays the following [30]:

- Hardware-level performance information (such as instruction execution rates, cache utilization, processor element utilization, delays due to branches and interlocks) if available,

- Operating-system-level activity (such as context switches, address-space activity, system calls and interrupts, kernel performance statistics),
- Communication-library-level activity (such as message-passing, inter-processor communication),
- Language-runtime activity (such as parallel-loop scheduling, dynamic memory allocation), and
- Application-level-activity (such as algorithm phase transitions, execution time profiles, data structure accesses).

Jinsight [47], another program visualization tool developed by IBM Research, helped in visualizing and analyzing the execution of Java programs. It was useful for performance analysis, memory leak diagnosis, debugging, or other tasks which required better understanding of the program. Jinsight could explore many aspects of programs:

- Visualization - Visualizations helped in understanding object usage, garbage collection and sequence of activity in each thread from an object-oriented perspective.
- Patterns - Pattern visualizations helped in the analysis of large amounts of information in a concise form by extracting essential structure in repetitive calling sequences and complex data structures.
- Information exploration - Allowed filtering options, detailed exploration on different views and customized units that match features under study.
- Measurement - Enabled measurement of execution activity or memory summarized at any level of detail, along call paths, and along both these dimensions simultaneously.

- Memory leak diagnosis - Special features were provided to help diagnose memory leaks.

Both PV and Jinsight helped users debug their programs, perform performance analysis/tuning and understand the structure and dynamics of large object-oriented applications, frameworks, and libraries. Further, both the tools made use of color to show correlations between dynamic visualizations and hierarchal source code representations. Figure 2.7 shows several visualizations in PV, including a representation of the control flow of the program, a list of processes, and a display of system state and kernel statistics. Figure 2.8 shows a screenshot of Jinsight demonstrating its pattern-visualization capabilities.

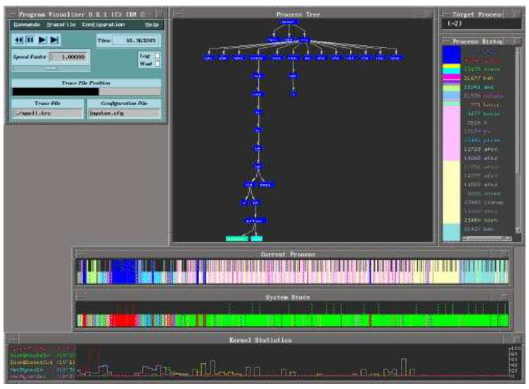


Figure 2.7: PV visualizing system statistics including the control flow, a list of processes, and a display of system state and kernel statistics [44]

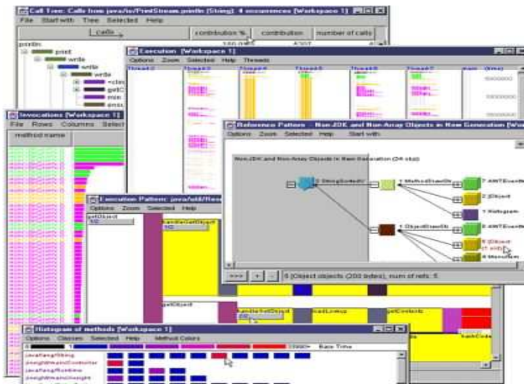


Figure 2.8: A screenshot of Jinsight showing use of color for effective pattern-visualization.

Java Visualizer, a dynamic software visualization system developed by Reiss [46], presented a view of programs in action. The system aimed at minimizing the overhead involved with the visualizations while maximizing the information. Figure 2.9 displays the class and package usage information on the left along with the status of threads in the system on the right. Box display visualization is used on the left half of the figure to display each class or package; height reflects the log of the number of entries to the class and width reflects the number of allocations done by the class or package. The darker displayed rectangle indicates where synchronized calls occur and the red rectangles indicate what is primarily being allocated at this point in the execution. The right half of the display uses additional box displays to indicate the status of the two threads in the system. The green box on the left (reading "main") indicates that the thread is actively running while the magenta box on the right (reading "Reference Handler") indicates the thread is waiting for an event.

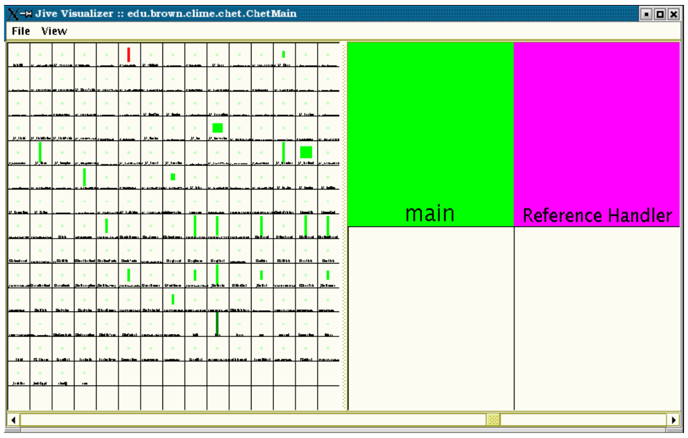


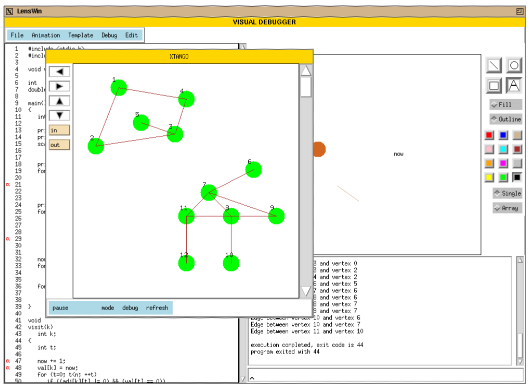
Figure 2.9: Java visualizer : Dynamic display of a Java program in action.

Java Visualizer did visualization for software understanding and focussed on maximizing the display information. It used IBMs Jikes Bytecode Toolkit [23] to instrument the Java programs in order to insert calls wherever significant events occurred, as a result of which the application had a considerable startup overhead.

2.3 Debugging

Software Visualization has been shown to aid debugging [53, 2, 57, 37, 36]. Jinsight [26] was used for debugging and optimization. ZStep95 [34] provided visualizations of the correspondences between static program code and dynamic program execution for the purpose of debugging. LENS [37] allowed programmers to create algorithm animation style presentations for the programs they were debugging. The screenshot seen in Figure 2.10 shows LENS system being used to examine a depth first search program. On the top is the generated animation of the program. In the lower window, to the left is the program source, in the top right is the graphical editor for specifying the appearance of the program, and in the lower right is the debugger command shell [32].

Though it may seem that most of the development in the field of Software Visualization has dealt with procedural languages, one of the most successful automatic systems was developed as a graphical tracer and debugger for the declarative language Prolog. Transparent Prolog Machine (TPM) [17], itself written in Prolog, provided an execution model and graphical debugger for logic programming. TPM presented two basic views to the user: the coarse-grained view and the fine-grained view. The coarse-grained view (CGV) is based on a traditional and/or tree; it shows the execution space of the entire program with nodes representing goals. The fine-grained view (called AORTA) allows the user to zoom in on a particular node to get details of data flow, such as variable instantiation. Figure 2.11 displays the coarse-grained view of a query paused in the middle of a replay. The tool palette seen on the left includes 6 video replay buttons, and tools for pausing, refocussing, accessing other windows, and zooming [42].



```
void
visit(k)
   int k;
{
   int t;

   now += 1;
   val[k] = now;
   for (t=0; t<n; ++t)
```

```
Edge between vertex 10 and vertex 7
Edge between vertex 11 and vertex 10

execution completed, exit code is 44
program exited with 44
```

Figure 2.10: LENS tool examining a depth first search program [32]

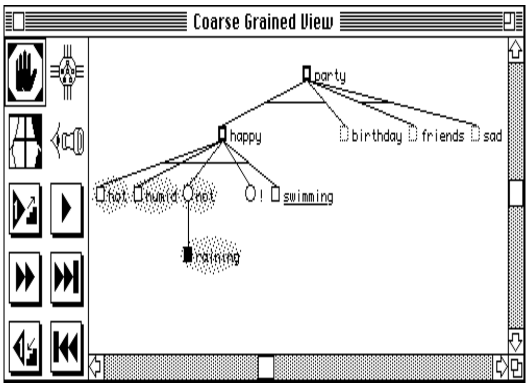


Figure 2.11: Screenshot of TPM displaying an example of CGV and the basic control panel [42]

2.4 Integration

Attempts have been made in the past to integrate visualization systems with development environments, the most prominent one being plugging-in the SHriMP Views visualization tool [54], described in [35]. SHriMP presents hierarchically structured information using a nested graph view. It introduces the concept of nested interchangeable views to allow a user to explore multiple perspectives of information at different levels of abstraction [54]. [35] presents a work strikingly similar to ours, where the authors describe their experiences integrating an existing, stand-alone software understanding tool, SHriMP, into the Eclipse development environment. Further the authors detail the issues they faced while integration and provide recommendations for other developers trying to integrate visualization systems into Eclipse.

2.5 Summary

This chapter explored the related work done in the field of Software Visualization. We highlighted the work motivating the use of visualization for program understanding and debugging. We explored various categories of software visualization and presented many visualization tools. Towards the end, we focussed on visualization tools for debugging and presented a recent work aimed at integrating a visualization system with a well known development environment.

Chapter 3

System Enhancement

An evaluation of the original SoftViz system brought forward certain limitations of the system, described in section 1.2. Various enhancements have been made in this thesis not only to remove these limitations but also to add new functionalities. The focus of this chapter is to present the work done on the system to improve its functional characteristics as well as efforts put into its plug-ins. Section 3.1 describes the enhancements made to the system core and section 3.2 presents the work done on the plug-ins of SoftViz.

3.1 Enhancement of core functionality

Enhancements made to core functionalities of SoftViz were aimed at removing bottlenecks in the previous design. New features such as stream buffering were included to streamline the operation and expand the capabilities of the system. Four major improvements are described below.

3.1.1 Stream storage and retrieval

Stored stream handling was proposed to provide a feature wherein a live (real-time) event stream could be stored to disk and played back later, to produce exactly the same visualization results. Adding persistence to event streams was accomplished by creating two files: the data (.dat) file and the index (.idx) file. The data file contains actual events, whereas the index file contains pointers to the events in the data file, indexed by event timestamps. However the implementation of this architecture was incomplete and inaccurate. The index files were unstructured which made it difficult to access the file randomly, to reach the nth event entry one had to traverse through each of the previous (n-1) events.

The storage of live event streams and retrieval of stored streams for playback has now been completely implemented. We added structure to the index files that enabled random access, making it possible to perform advanced stream control operations. The operations supported by the stored stream architecture are listed below:

- Play/Stop stream
- Pause/Resume playback
- Asynchronous access to “previous” and “next” events
- Control the pace of visualization

The new implementation made these operations feasible, without causing much overhead. These operations were made a part of the stored stream manipulation directives, so that the user could control the visualization.

3.1.2 VCR-like stream control panel

The requirement was to provide the user with an interface to control the parameters of a stored stream playback. The features provided by stored stream playback included playing the stream, stopping the playback, pausing and resuming, accessing the previous and next events and varying the pace of visualization. The desire was to provide control of these features to the user through a simple, intuitive user interface.

An interface resembling a Video Cassette Recorder (VCR) control panel was chosen due to its simplicity and a striking analogy between operating a VCR and controlling the visualization of an event stream. The user could play/stop/pause/resume the playback of an event stream just like a movie, fast forward (access next event) and rewind (access previous event) and even vary the pace to speed up or slow down the visualization. The visualization could be played at multiples of the original pace, for example twice or thrice the normal for speeded up display and half or one third of the same for slow motion effect. This feature is available only for stored streams, wherein the pace of the event stream could be controlled by varying the timing between two events. The thread extracting the information from the files was configured to control the timing; this produced an effect of slow motion or speeded up visualization, the pace of which could be controlled by the user.

The resemblance to a well known electronic system (VCR) made it easy for the user to understand and operate the controls without the requirement of any formal specifications. Figure 3.1 shows the control panel, along with descriptions of various controls.

3.1.3 Stream Buffering

The original design of the system did not have any provision for buffering of events. The visualization plug-ins were either too simple to require information about past events (such as Sunburst) or they had to do event buffering within themselves (such as Time-line). As more intelligent and versatile plug-ins were developed, they required access to past information. Also, maintaining a separate event buffer within each plug-in raised scalability concerns, especially considering the fact that even small sized programs emit-

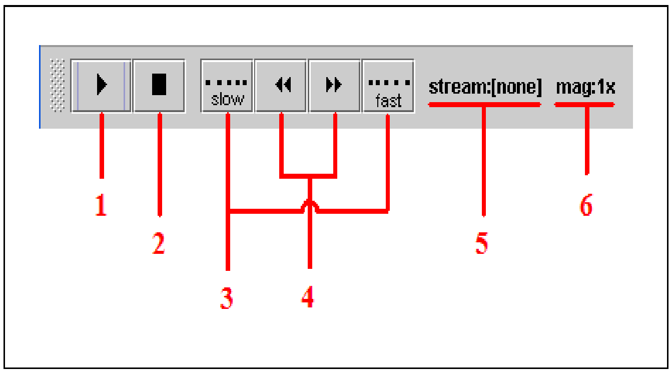


Figure 3.1: VCR-like event stream control panel. 1) Play/Pause/Resume button 2) Stop button 3) Slow down/speed up playback 4) Access previous/next event 5) The stream being visualized 6) Magnification factor.

ted thousands of events. What was required was a central event buffering mechanism to which each plug-in can query for past events.

A buffering utility was added to the SoftViz core that buffers the events received from a live or stored event stream. This buffer implements a “Cursor” interface through which the plug-ins can request past events [45]. Note that the plug-ins receive the current events as before; plug-ins post a request to the buffer only if they need access to past information. As an outcome of this design, plug-ins such as Sunburst that do not require access to past events never need to query the buffer. The utility buffers the events in a temporary file on the file system, a file that is created only when the buffer is instantiated and deleted as soon as the buffer is destroyed. The choice of buffering in a file rather than in memory (in some structure such as a Vector or ArrayList) was made to maintain robustness of the system in a scenario where tens of thousands of events are received (in which case an in-memory buffer might break).

The buffering utility is, by design, thread-safe. This was not only desired but required since the utility handles writing to the buffer concurrently with reading from it to satisfy requests by one or more plug-ins. Since events are only appended to the end of the file, concurrent access is safe; a classic example of “One writer multiple readers” paradigm.

Further, the request for past events can be made w.r.t timestamps or an event sequence number, meaning that a plug-in may request events between any two timestamps t1 and t2, or between any two sequence numbers n1 and n2. This lays a foundation for a flexible request criteria providing the plug-ins freedom to request events within a window over all past events, where the size and location of the window is as specified by the plug-in.

3.1.4 Enhancement of event structure

Another significant improvement occurred in the event structure of SoftViz. The original design provided only five fields: timestamp, class name, method name, interface name and thread ID. This structure was too constrained to be of use for advanced plug-ins that may require additional information, such as method parameters. Further, it was evident that some optimization may be achieved by using codes for class names and method names to avoid expensive string compare operations. Also, event sequence information was desirable. In light of the requirements stated above, new fields containing the following information were added to the structure:

- Event sequence number: Signifies the sequence number of this event. Sequence numbers start from 0 increasing by 1 as each event is received.
- Class code: A code that uniquely identifies the class to which this event belongs. The code is simply an integer; the first class is assigned a code 1 and the rest are assigned codes incrementally.
- Method code: A code that uniquely identifies the method to which this event belongs. Methods are assigned codes in a manner similar to classes.
- Parameter count: A number signifying the number of parameters in the method to which this event belongs. It is zero for events not associated with methods.

- Parameter list: If the Parameter Count is greater than 0, this is the list of parameters. Parameters are stored as strings.

The enhanced event structure not only provides more information about the event itself, but also makes room for optimizations and more flexible processing of events. It should be noted that the codes (class as well as method codes) are assigned by the buffering utility, and this information would not be available in the current events received directly by the plug-ins.

3.2 Enhancements to plug-in architecture

The two existing plug-ins of SoftViz were only partially implemented; in particular plug-in Timeline required a lot of work. The enhancements made to the plug-ins are described below.

3.2.1 Sunburst

According to [31] “Sunburst visualization represents a class of visualizations capable of representing a hierarchical data set in an intuitive and space-efficient manner.”

Introduction

Sunburst visualization provided both the static information about the system by representing the hierarchical organization of system structure, as well as dynamic information by modeling the control flow. Sunburst represented structure of the system in a radial display of segments of concentric circles [31], the innermost circle representing the root of the hierarchy. Animation was used to convey the dynamic information; brightness and saturation of a segment of the circle were varied to indicate that segment in execution. Figure 3.2 shows a snapshot of Sunburst in action.

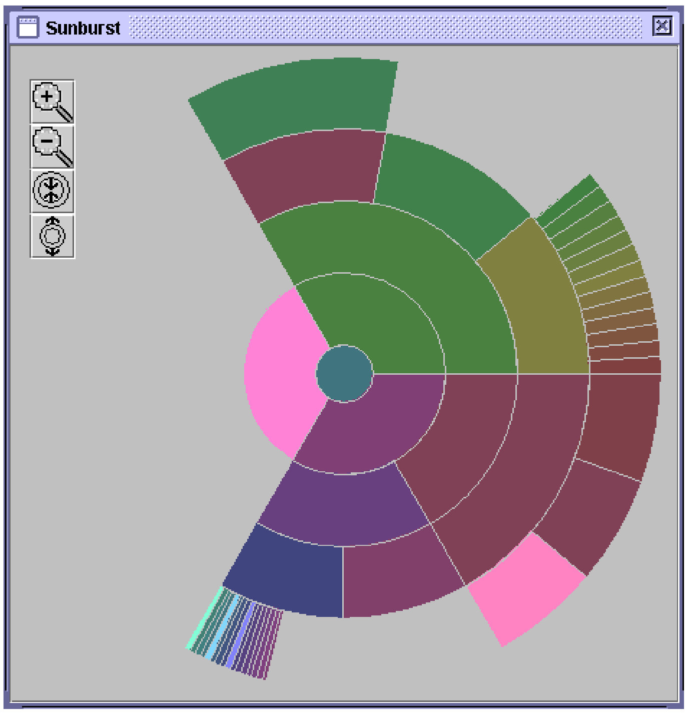


Figure 3.2: The Sunburst Visualization.

Enhancement to Sunburst

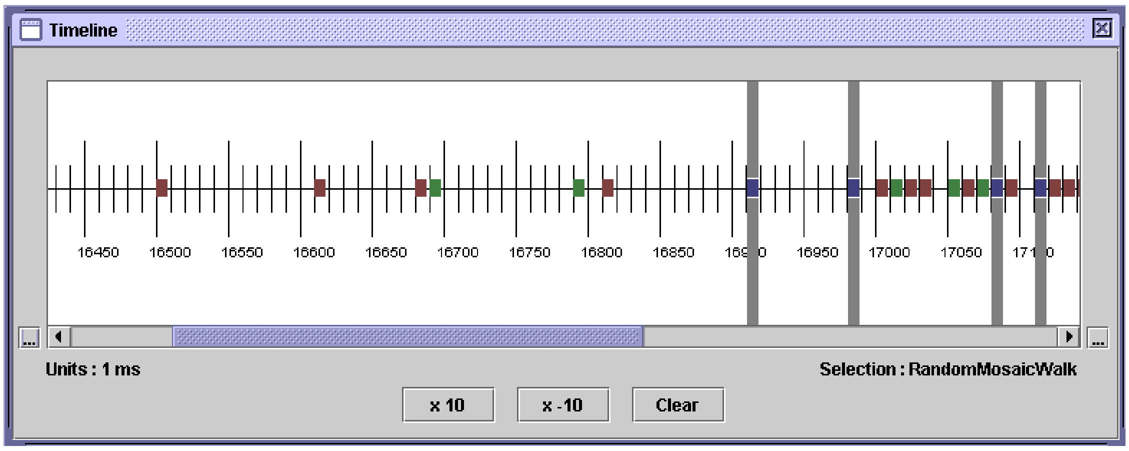
Sunburst's zooming/unzooming facility was not implemented. We used an intelligent algorithm combining linear and non-linear scaling to implement the desired scaling feature. The circular structure of Sunburst visualization made linear scaling ineffective at low scale values, whereas the non-linear scaling algorithm caused too large steps at larger scale values. The best of both worlds was achieved when the two techniques were combined with an appropriate threshold to use non-linear scaling at lower scale values and linear scaling at medium and larger scale values.

3.2.2 Timeline

Timeline visualization is representative of a set of visualizations that display dynamic information referenced by time, depicting temporal relationship between various events.

Introduction

The Timeline visualization is based on dynamic behavior of the system, where the events are placed on a timeline [31] to reveal temporal relationships between different events. Information about the source of the event (the class from which it originated) is displayed by coloring the bar displaying the event according to the color-map set by SoftViz. Timeline provides the temporal sequencing of events and gives some idea about the extent of separation of events in time (by depicting time between events). Unlike Sunburst, Timeline does not provide any static information about the system. Figure 3.3 shows a snapshot of the Timeline visualization.



Units : 1 ms

Selection : RandomMosaicWalk

Figure 3.3: The Timeline Visualization

Enhancements to Timeline

Implementation of Timeline was incomplete. There were no controls for abstraction and scaling. The work on Timeline began with implementation of a scroll-bar to view all events within the plug-in window. To view events beyond the window, a paging utility was designed, which resembled placing a window over a time-line of infinite length. Note the usage of a lower case, dashed word, ’time-line’ instead of ’Timeline’; ’Timeline’ refers to the plug-in as a whole, whereas ’time-line’ refers to the representation of time on a graph within the plug-in. The user can move the window over to a different portion of the time-line to view events in that portion. The combination of the scroll-bar (to view events within the window) and paging utility (to view events over multiple windows) allows viewing of all the events over the entire virtual length of time. Scaling was also implemented, which allows users to view the time-line at different levels of abstraction. The granularity of the scaling algorithm was 10 ms; the scale could be changed in steps of ten.

The events displayed on Timeline were color-coded as per their class names. This allows users to recognize the class of an event and differentiate events from different classes by simply looking at the color of the event. More work was done to allow users to select events from the time-line and to indicate the selection status of events (whether selected or not) to the user. Users can select a single event by clicking on it. Selection of multiple events is provided by either selecting the desired events while pressing the “Ctrl” key, or by simply clicking and dragging over the area where events of interest lay. In case of single event selection, the class name of the selected event is displayed in the status bar just below the window containing the time-line. Selection of event(s) is depicted by dark bars above and below the plug-in.

Various other modifications were made such as enhancement of the GUI, implementation of efficient event selection procedures, and a centering capability wherein right-clicking anywhere on the time-line made the window center at the clicked position. A snapshot of the current Timeline visualization is displayed in Figure 3.3.

3.2.3 Interaction between plug-ins

The plug-in based design of SoftViz provided a scope for allowing different plug-ins to collaborate. The original design of SoftViz provided some incentives in this direction by recommending a static conformance on the plugins, however a much more useful and effective dynamic interaction standard was proposed and developed as a part of this work. Both the techniques are described below:

Static conformance

The original design of SoftViz recommended a static conformance on its plug-ins. The system first assigned colors to each class being visualized, and then passed on this map of class-color pairs to each of its plug-ins. The recommendation was that each plug-in use the same color codes for visualizing the class information, so that the user could better relate information from different plug-ins. As mentioned, this was only a recommendation and the plug-ins could choose to implement a different coloring strategy.

Dynamic Inter-plug-in communication

The static conformance proved helpful but a deeper level of interaction was required. An interaction standard was required that could adapt to the current state of the visualizations and respond to the dynamic nature of the behavior. The inter-plug-in communication channel was proposed as a solution to this requirement. The communication channel was designed and implemented to allow the plug-ins to dynamically exchange special events within themselves, at run-time. These special events are treated separately from the events obtained through the event stream. Three special events have been defined in the system:

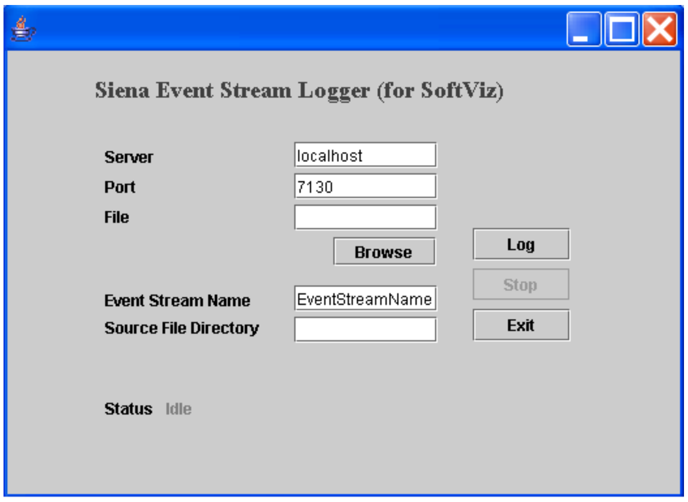
- Select - Indicates selection of a particular “event stream event”,
- Unselect - Indicates de-selection of the specified “event stream event”, and
- Clear - Indicates de-selecting all previously selected events.

The communication channel was designed and developed with an aim of being able to “group” certain visualization plug-ins, allowing them to communicate internally to collaborate efforts and come up with a re-enforced aggregate view of the system. The premise was that two plug-ins working in concert would yield more than the two working individually without any collaboration. The channel can be developed further using more and better event primitives so that a greater level of cooperation is achieved between the plug-ins. It is believed that as more intelligent plug-ins are developed, the communication channel will be developed to gain advantage of the combined effort of two or more plug-ins.

3.3 SIENA based Event Logger

A small and simple stand-alone utility, Logger, was created to be able to write log files (stored stream files) without invoking SoftViz. The idea was to create many log files using Logger so that they could be analyzed at a later time. Event streams stored in log files could be loaded up in SoftViz and played-back at any time, and the visualizations would behave as if the events were happening in real-time. Visualizing stored streams presents other advantages as well, such as the ability to control the event stream by causing it to play/pause and stop as desired. Stored streams also allow the last event to be replayed and the next event to be invoked before its scheduled time. Further, stored streams allow the user to control the speed of the visualization to suit his/her needs.

Even though Logger was a separate application, it's interface was kept similar to that of SoftViz. As can be seen in a screenshot of Logger shown in Figure 3.4, it needed the same parameters as required if using the SoftViz system for creating stored stream. Once all the information was entered, the user could press "Log" button to start the logging of events. The user could press the "Stop" button to prematurely terminate the logging process. It created the stored stream that consisted of two log files (.idx and .dat files) in the user's working directory.


Siena Event Stream Logger (for SoftViz)

Server	localhost
Port	7130
File	
Event Stream Name	EventStreamName
Source File Directory	

Status Idle

Figure 3.4: Event Logger for SoftViz.

The Logger uses SoftViz's SIENA library and packages involved in stream storage to extract events and store them into files. A disadvantage of Logger is that it can work only with the SIENA based mode of operation; it does not scale to use other techniques of event extraction such as the JPDA based Trace utility now included in SoftViz's repertoire of event extraction techniques.

3.4 Summary

This chapter presented the enhancements made to SoftViz, both to the framework and its plug-ins. Enhancements made to the framework included completion of stream storage and retrieval functionality and implementation of the VCR-like stream control panel and the event stream buffer. Next we introduced the two plug-ins, Sunburst and Timeline, and listed the enhancements made to them. Finally we discussed the design of the inter-plug-in communication channel and introduced the "Logger" tool developed to create stored streams.

Chapter 4

Integrated Environment

SoftViz was designed as a stand-alone system, and relied on code-intrusive methods of information extraction. This design does not lend itself well to debugging since the debugging tools are expected to be more closely related to the application being debugged. Further this design necessitated the use of some form of middleware to act as a bridge between Softviz and the target application being visualized. The SIENA event distribution bus [48] bridged this gap in the original implementation of SoftViz, but it posed performance problems as it was slow and had considerable overhead [31].

Various design alternatives were considered to solve the problems posed by the stand-alone design of SoftViz. Section 4.1.2 summarizes the design alternatives and their advantages and shortcomings. As explained in section 4.1.2, it was found that integration of SoftViz with the Eclipse development environment [15] had the most advantages. Eclipse is a general purpose platform upon which other tools can be built as plug-ins. Many other individuals and commercial and research groups have developed plug-ins for the Eclipse platform, such as profiling tools, debugging tools, and version control tools, amongst others. SoftViz now exists as an Eclipse plug-in and can be invoked at a simple press of a button from the development environment. The integration is shown in Figure 4.1.

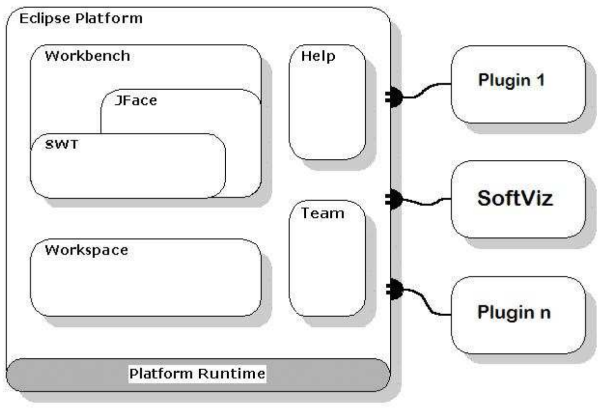


Figure 4.1: Integration with Eclipse

We decided to use Java Platform Debugger Architecture (JPDA) [25] to extract events from the target application for the new SoftViz architecture. JPDA provides the infrastructure to build debugger applications for the Java 2 Platform, Standard Edition (J2SE). It includes three-layered APIs: Java Debug Interface (JDI), a high-level Java programming language interface that includes support for remote debugging; Java Debug Wire Protocol (JDWP), which defines the format of information and requests transferred between the debugging process and the debugger front end; and Java Virtual Machine Debug Interface (JVMDI), a low-level native interface that defines the services a Java virtual machine must provide for debugging. This brings forth two advantages, first being that SoftViz no longer depends on the heavy-weight SIENA bus, which was formerly the sole mechanism for event delivery. Though no formal evaluation has been done to compare the speeds of the JPDA based system with the old SIENA based one, the visualizations produced by JPDA based system are perceptively faster. A second advantage is gained by the usage of non code-intrusive techniques for event extraction, which unlike the earlier technique, does not require any manipulation of the source code.

4.1 Design

This section introduces the new SoftViz design and outlines all the design alternatives considered. The first subsection brings forth the design of SoftViz as as Eclipse plug-in and provides an overview of different modules comprising the new architecture. The second subsection lists various design alternatives considered and their limitations leading to reasons for not pursuing them.

4.1.1 SoftViz as an Eclipse plug-in

The new architecture of SoftViz consists of three main modules, as shown in Figure 4.2. The figure also displays the control flow and interactions between the various modules.

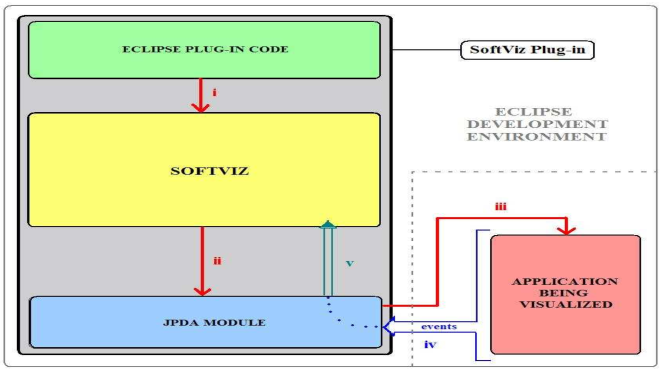


Figure 4.2: Architecture of SoftViz. Invocation sequence also shown. (i) SoftViz invoked; (ii) SoftViz invokes JPDA module; (iii) JPDA module launches the application to be visualized; (iv) Events from the application flow into JPDA module; (v) Events forwarded to SoftViz.

SoftViz is comprised of three modules:

- Eclipse plug-in code - The Eclipse plug-in code initializes the plug-in, and is generally found in all Eclipse plug-ins. This code is mostly generated by Eclipse automatically while the plug-in is being developed.

- SoftViz - This is the modified SoftViz system originally developed as a stand-alone application. The main modifications from the perspective of integration with Eclipse include processing of events received from the JPDA module and integration of this functionality with other technologies such as buffering and stream storage.

- JPDA module - This module configures the virtual machine in debug mode and launches the application to be visualized. It receives the events from the application and forwards them to SoftViz. This module is based on "Trace" [56], an example of the JPDA technology released by Sun Java [55]. This module uses the Java Debug Interface (JDI). Further, the programs need not be compiled before debugging since events are extracted from the Java Virtual Machine (JVM) and no extra information is required for generating the events.

The execution sequence is as follows:

- STEP 1: The user invokes the SoftViz plug-in from the development environment. This causes the plug-in to be loaded and the "Eclipse plug-in code" module to be executed. This module instantiates the JPDA module, and then invokes SoftViz.

- STEP 2: SoftViz initializes itself and displays options on the screen. Once the user chooses the "launch" mode and fills in the required information, SoftViz invokes the JPDA module.

- STEP 3: The JPDA module configures the Java Virtual Machine (JVM) and launches the target application.

- STEP 4: JVM starts delivering the requested events to the JPDA module.

- STEP 5: JPDA module forwards the events to SoftViz for further processing and visualization.

SoftViz, when invoked in "launch" mode, circumvents the need of the SIENA event distribution bus. However SoftViz can still be run in a pseudo-stand-alone mode within Eclipse, though SIENA would be required for this mode of operation.

4.1.2 Design alternatives

There were various design alternatives considered before the design described in subsection 4.1.1 was chosen. The first design alternative was to maintain SoftViz as a stand-alone system, while substituting SIENA [48] with a more efficient event transport system. However this would have solved only a part of the problem, since this approach would have maintained the dependence of SoftViz on the code-intrusive technique of event extraction. A better option was to use some module such as JPDA [25] for code-independent extraction of information. This was an attractive option, though it was not enough by itself; some kind of integration was still required to enable close interaction of SoftViz with the applications being debugged. Hence, integration of SoftViz with some development environment while eliminating the need for SIENA was the best solution.

SoftViz could have been integrated with open source systems, such as debuggers such as gdb [20] or development environments such as BlueJ [5] and Eclipse [15]. However, of all the options, integration with Eclipse was most desirable and most practical, due to the following reasons:

- The Eclipse platform has been designed for integration of tools in the form of plug-ins.
- Eclipse is a widely used development environment.
- Integration with Eclipse provides scope for wide distribution of the tool.

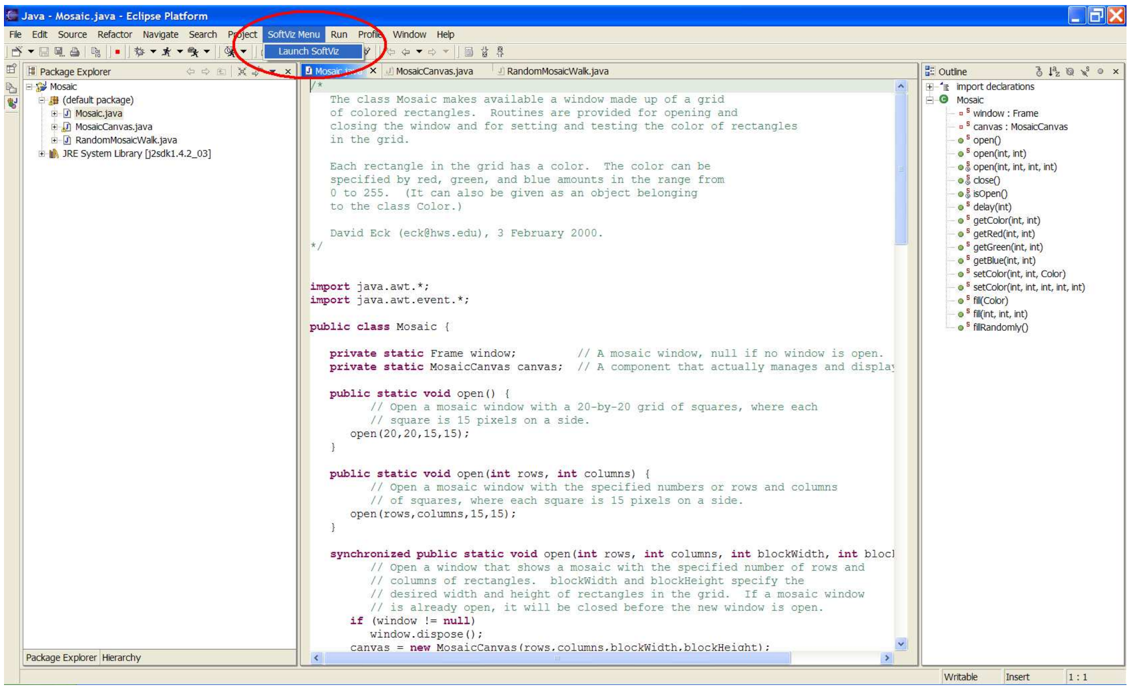
Integration with Eclipse is possible at various levels, and the level of integration depends on specific needs and time constraints. Integration of SoftViz with Eclipse is very light (shallow integration) as there is no integration of SoftViz with other Eclipse-based tools. As a matter of fact, the article “Levels of Integration” [33] defines five levels of integration:

1. None - No integration, tools are separate and independent,
2. Invocation - Integration is through invocation of registered applications on resource types,
3. Data - Integration is achieved through data sharing,
4. API - Tools interact with other tools through Java APIs that abstract tool behavior and make it publicly available, and
5. UI - Tools and their user interfaces are dynamically integrated at runtime including window panes, menus, toolbars, properties and preferences.

In light of the above categorization, the integration of SoftViz with Eclipse falls in the first category, “None”. SoftViz and other Eclipse tools are totally independent. There are various reasons for this:

1. The original stand-alone design of SoftViz meant that it is inherently self-sufficient. It has its own GUIs and control panels, and forcing SoftViz into a deeper level of integration would have required a complete re-design of the system.
2. Given the requirements and capabilities of SoftViz, there are no visible benefits of deeper integration at this point of time.
3. There was an endeavor to integrate SoftViz at the second level, “invocation”, but it was unsuccessful. The effort fell into problems with invoking the target application within Eclipse and generating the events of interest. This remains a future work.

4.2 Screenshots

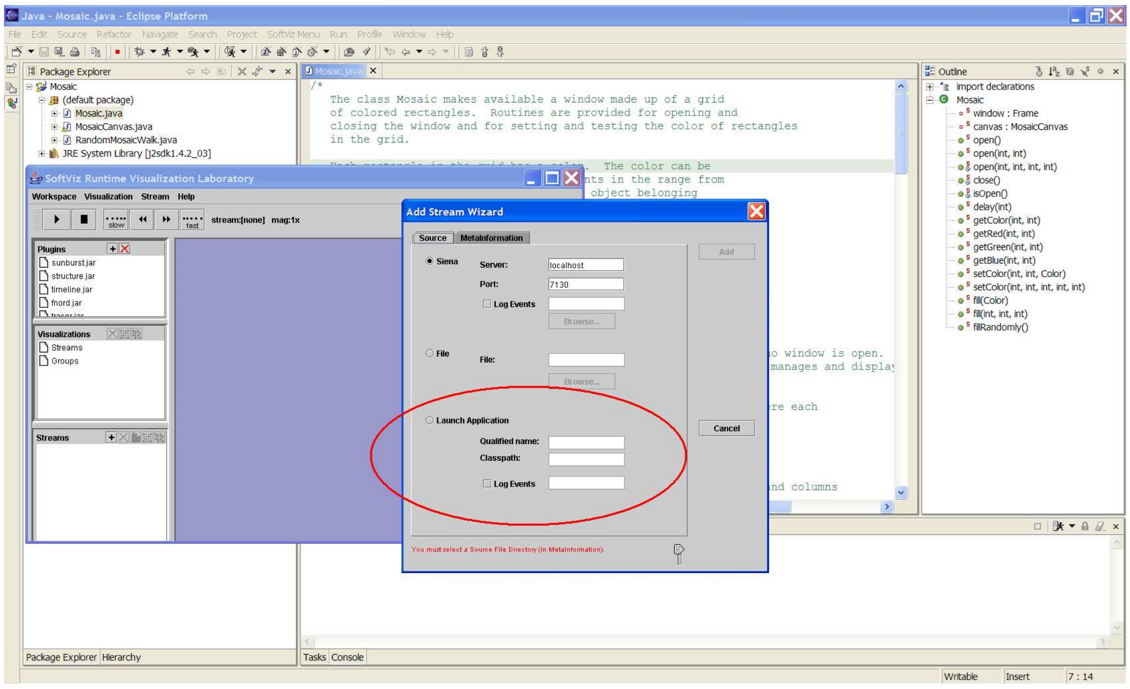


```
/*
   The class Mosaic makes available a window made up of a grid
   of colored rectangles.  Routines are provided for opening and
   closing the window and for setting and testing the color of rectangles
   in the grid.

   Each rectangle in the grid has a color.  The color can be
   specified by red, green, and blue amounts in the range from
   0 to 255.  (It can also be given as an object belonging
   to the class Color.)

   David Eck (eck@hws.edu), 3 February 2000.
*/


import java.awt.*;
import java.awt.event.*;

public class Mosaic {

   private static Frame window;         // A mosaic window, null if no window is open.
   private static MosaicCanvas canvas;  // A component that actually manages and display

   public static void open() {
         // Open a mosaic window with a 20-by-20 grid of squares, where each
         // square is 15 pixels on a side.
      open(20,20,15,15);
   }

   public static void open(int rows, int columns) {
         // Open a mosaic window with the specified numbers or rows and columns
         // of squares, where each square is 15 pixels on a side.
      open(rows,columns,15,15);
   }

   synchronized public static void open(int rows, int columns, int blockWidth, int blocl
         // Open a window that shows a mosaic with the specified number of rows and
         // columns of rectangles.  blockWidth and blockHeight specify the
         // desired width and height of rectangles in the grid.  If a mosaic window
         // is already open, it will be closed before the new window is open.
      if (window != null)
         window.dispose();
      canvas = new MosaicCanvas(rows,columns,blockWidth,blockHeight);
```

Figure 4.3: Screenshot showing Eclipse IDE

Figure 4.3 displays a screenshot of the Eclipse IDE showing the run-time environment, an environment used for development and deployment of Eclipse plug-ins. As can be seen in the package explorer, a project named "Mosaic" has been chosen to be visualized. Mosaic is a simple open source Java program obtained from [14]. Mosaic draws a grid of colored rectangles on the screen. The circle highlights the SoftViz plug-in menu, used to invoke SoftViz. The menu is used to launch SoftViz, which then provides options for launching the desired application.

The screenshot displayed in Figure 4.4 was captured after SoftViz was loaded, and the user invoked SoftViz's "Add Stream Wizard". This wizard controls SoftViz's mode of operation, allowing the user to select SIENA stream, stored stream or the JPDA based Trace event stream. The highlighted circle focuses on the updated portion of the wizard,



```
/*
   The class Mosaic makes available a window made up of a grid
   of colored rectangles.  Routines are provided for opening and
   closing the window and for setting and testing the color of rectangles
   in the grid.
```

(•) Siena
Server: localhost
Port: 7130
☐ Log Events

( ) File
File:

( ) Launch Application
Qualified name:
Classpath:
☐ Log Events

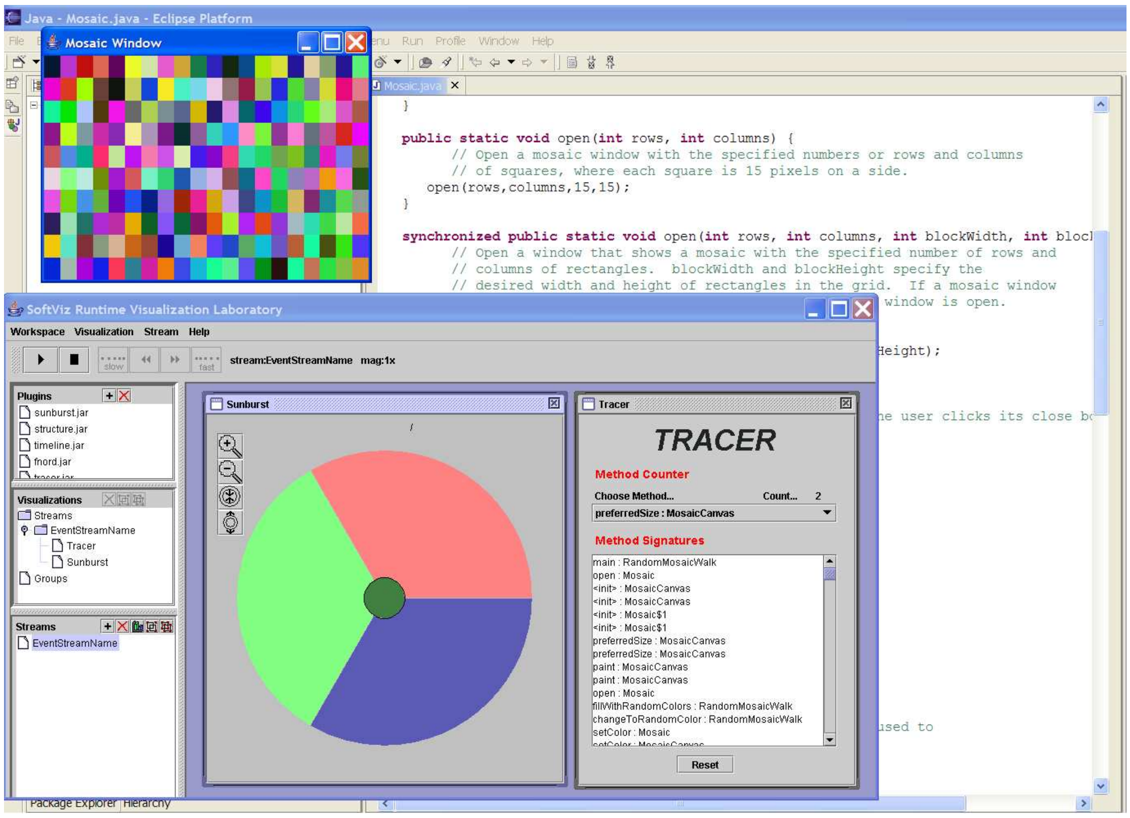
You must select a Source File Directory (in MetaInformation).

Figure 4.4: Screenshot showing SoftViz's "Add Stream Wizard".

which lets the user launch the target application and visualize the events extracted by the Trace module. This option requires the user to enter information used to launch the target application including the application's fully qualified name and its classpath. Like the other options, an absolute path to the application's source code is required, used to generate the color-map.

Figure 4.5 depicts SoftViz visualizing a target application. The application (Mosaic) can be seen running on the top left. Two visualization plug-ins, Sunburst and Tracer, are being used to visualize Mosaic. As can be seen, Sunburst is visualizing class accesses by varying the brightness of segments representing the respective classes. Also Tracer is displaying the execution footprint and keeping a count of each unique access at the same time.

Figure 4.6 displays SoftViz visualizing the XmdvLite, a lightweight (Java) version of the XmdvTool [58]. XmdvTool (webpage [59]) is a public-domain software package



```
}

public static void open(int rows, int columns) {
      // Open a mosaic window with the specified numbers or rows and columns
      // of squares, where each square is 15 pixels on a side.
   open(rows,columns,15,15);
}

synchronized public static void open(int rows, int columns, int blockWidth, int bloc
      // Open a window that shows a mosaic with the specified number of rows and
      // columns of rectangles.  blockWidth and blockHeight specify the
      // desired width and height of rectangles in the grid.  If a mosaic window
                                                            window is open.

                                                           Height);

                                                           he user clicks its close b

                                                           sed to
```

```
Choose Method...                Count...   2
preferredSize : MosaicCanvas

main : RandomMosaicWalk
open : Mosaic
<init> : MosaicCanvas
<init> : MosaicCanvas
<init> : Mosaic$1
<init> : Mosaic$1
preferredSize : MosaicCanvas
preferredSize : MosaicCanvas
paint : MosaicCanvas
paint : MosaicCanvas
open : Mosaic
fillWithRandomColors : RandomMosaicWalk
changeToRandomColor : RandomMosaicWalk
setColor : Mosaic
```

Figure 4.5: Screenshot showing SoftViz working within Eclipse.

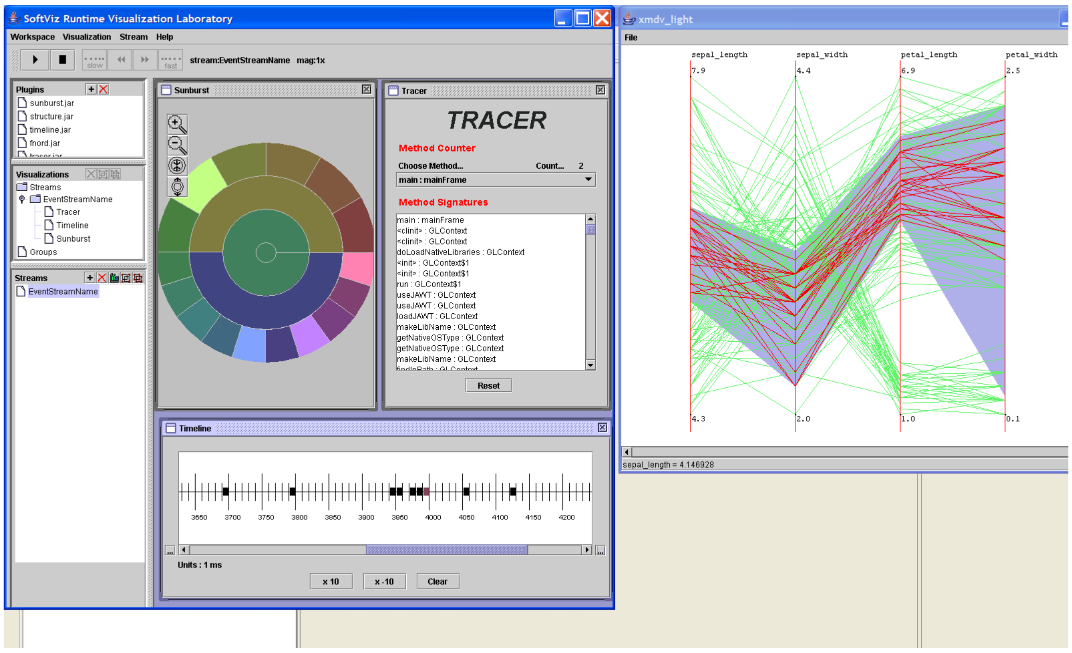
for the interactive visual exploration of multivariate data sets. XmdvTool was created by Ward [58] in 1994, and the lightweight Java version “XmdvLite” was released in 2003.

4.3 Advantages of Integration of SoftViz

Integration of SoftViz will increase the accessibility of the system and thus encourage greater usage of the system by more and more people. We believe integration with Eclipse improves the efficiency of the system by circumventing the usage of the slow and inefficient SIENA event distribution bus. This integration provides the following advantages:

1. Ability to extract events using techniques that are not code invasive .

2. Ability to specify the set of events of interest on the fly.



```
main : mainFrame
<clinit> : GLContext
<clinit> : GLContext
doLoadNativeLibraries : GLContext
<init> : GLContext$1
<init> : GLContext$1
run : GLContext$1
useJAWT : GLContext
useJAWT : GLContext
loadJAWT : GLContext
makeLibName : GLContext
getNativeOSType : GLContext
getNativeOSType : GLContext
makeLibName : GLContext
```

Figure 4.6: SoftViz visualizing XmdvTool [58].

3. Better filtering options inducing minimal overhead.

4. Ability to access exceptions.

5. Lightweight operation.

6. Increased efficiency.

7. Potentially increased usage.

4.4 Implementation and Distribution

Implementation of SoftViz as an Eclipse plug-in was an interesting experience. SoftViz was originally designed and built as a stand-alone application and a deep integration with Eclipse would have required substantial changes in its design. To minimize the design changes, we chose to perform a high level (or shallow) integration. Eclipse's Plug-in

Development Environment (PDE) was helpful in getting started with development of the plug-in. Based on responsibility, the implementation was divided into three major modules, Plug-in module, SoftViz module and JPDA module, as described in Section 4.1.1.

SoftViz requires some libraries (jar files), and Eclipse plug-ins do not allow inclusion of jar files in the same way as other Eclipse projects do. A query was posted on Eclipse Newsgroup asking how to include jar files in Eclipse plug-ins. A quick response informed that another (resource) plug-in was required that would contain all the necessary jar files, and this plug-in would be used as a resource in the main plug-in. Hence was developed the “tools” plug-in, a resource plug-in for SoftViz. Therefore, integration of SoftViz into Eclipse requires two plug-ins, the “tools” plug-in containing SoftViz resources and the SoftViz plug-in itself.

The distribution includes both Eclipse plug-ins that comprise the SoftViz plug-in. The libraries that allow using SIENA event extraction mechanism are included within the plug-in, however the JPDA based event extraction implementation (also included) is recommended for extraction of events.

4.5 Summary

This chapter discussed one of the main contributions of the thesis: integration of SoftViz into Eclipse. We discussed the design of the integrated system and revealed various design alternatives considered, along with their advantages and shortcomings. Next we provided some screenshots of the system depicting the invocation and operation of SoftViz within Eclipse. Finally we outlined the advantages gained by integrating SoftViz into Eclipse.

Chapter 5

Error Classification Framework

In this thesis an Error Classification Framework has been developed that relates application types and associated error classes with the visualization options. Various classes of errors (as also identified by [16, 28, 22]) generally encountered within chosen application types have been explored, and associated with the visualization options that exist or could be developed to enable users to effectively detect them. The developed plug-ins have been classified based on their ability to enable users to detect certain errors that are representative of their respective error classes.

The theory and motivation behind our approach is the fault/error/defect model as specified in [11]. This model has been discussed earlier in Section 1.3.1, although for quick reference we define the three important terms again. The model defines a **fault** to be the observed deviation between software specification and its actual behavior. A fault may be triggered by one or more errors. **Error** is defined as the manifestation of defect at run-time, and **defect** is the design or coding mistake that may cause abnormal software behavior.

The visualization plug-ins display the behavior of the observed system. This display can correspond to a behavior that is normal (error-free) or faulty. It is up to the user to detect any aberration in the display to catch the manifestation of the error as soon as possible. However, the design of the plug-in can play a very important role. A well designed plug-in can allow the user to isolate the error and be a great help in detection. Finally the user has to find the exact location of the defect.

This would not only help the users to select the right visualization plug-in for the application at hand, but also assist in developing dedicated plug-ins that would allow the users to detect bugs of specific classes.

5.1 Framework Overview

The error classification framework presents common programming errors in eight categories. This primary categorization divides the errors depending upon their type (such as logic errors, data reference errors, interface errors). The errors within each category are further divided into secondary categories for an accurate grouping of related error types and separation of dissimilar ones. It should be noted that the error categories are not necessarily disjoint; there is some overlap. Figure 5.1 shows an overview of the error classification framework. The details and explanation of the categories follow in the next section.

5.1.1 Template

Details of each error categorization have been organized to follow a template described in this section. Each error class begins with a small introduction describing the error. The introduction is followed by:

- EXAMPLES: Examples of the error class are provided to establish a relation with the real world.

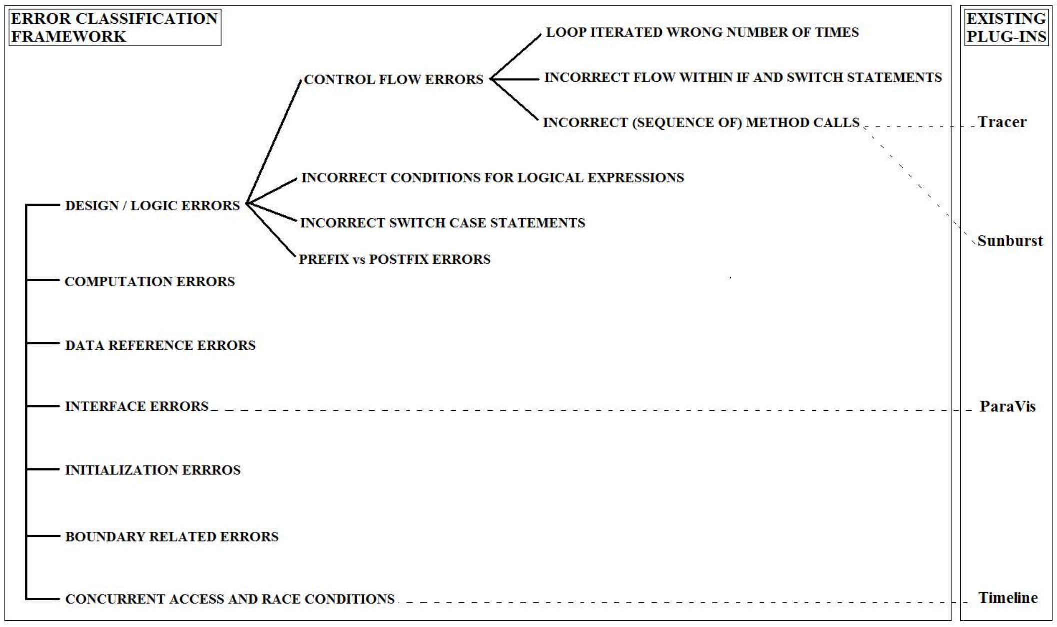

ERROR CLASSIFICATION FRAMEWORK
EXISTING PLUG-INS
LOOP ITERATED WRONG NUMBER OF TIMES
CONTROL FLOW ERRORS
INCORRECT FLOW WITHIN IF AND SWITCH STATEMENTS
INCORRECT (SEQUENCE OF) METHOD CALLS
Tracer
INCORRECT CONDITIONS FOR LOGICAL EXPRESSIONS
DESIGN / LOGIC ERRORS
INCORRECT SWITCH CASE STATEMENTS
Sunburst
PREFIX vs POSTFIX ERRORS
COMPUTATION ERRORS
DATA REFERENCE ERRORS
INTERFACE ERRORS
ParaVis
INITIALIZATION ERRROS
BOUNDARY RELATED ERRORS
CONCURRENT ACCESS AND RACE CONDITIONS
Timeline

Figure 5.1: Error Classification Framework overview and how existing visualizations fit

- MANIFESTATIONS: This section is included to give an idea how the error would manifest itself in an execution; this would help the users to relate a fault with the correct error category.

- DIFFICULTY AND FREQUENCY: Each error categorization is weighed on two independent dimensions, the frequency of occurrence and the difficulty of locating the defect causing the error. We believe this estimation will not only separate out the most troublesome error classes but also identify classes that require immediate work. It should be noted that this estimation is based on personal experience and may be questioned; it does not claim to be a formal yardstick of any kind.

- PLUG-IN DESIGN: This section describes a plug-in that would be helpful in detecting errors in this class. This description may be treated as a recommendation for designing dedicated plug-ins for detecting particular errors.

- PLUG-IN INPUT REQUIREMENTS: This description outlines the input requirements of the proposed plug-in.
- POSSIBLE DISPLAY: This provides a possible display scenario, sometimes accompanied by a figure detailing the “look and feel” of the plug-in.

5.2 The Error Classification Framework

The error classification framework categorizes programming errors into eight main categories. Each category is either described in its entirety or further divided into sub-categories that are then described individually.

5.2.1 Design/Logic Errors

Design/Logic errors are the errors that affect the sequence of execution of a program. These errors are generally caused due to defects rooted in the code that determines the flow of control [22], namely control constructs such as if-else constructs and loops. This error category has been identified in [22], [28] and [16] as well. According to [16], design/logic errors are among the top three most frequently occurring errors. This broad category of errors has been broken down into sub-categories as follows:

Control Flow Errors

Control flow errors is a huge category in itself, comprised of many errors. For this reason, control flow errors have been dealt with in a separate section.

Incorrect Conditions for Logical Expressions

Logical expressions determine the flow of execution based on a conditional clause, mostly within an if-else construct. Simple defects in the conditional expressions can alter the program flow and result in aberrant behavior. Further, the program may produce unexpected results only for particular cases, which makes these defects difficult to capture. These defects result in control flow errors as described in Section 5.2.2.

EXAMPLES: This category includes errors due to incorrect operators (e.g. $>$ instead of $>=$) [16], not equal versus equal when there are three cases, incorrect operand or operator, testing floating point values for equality, confusing inclusive and exclusive OR, incorrectly negating a logical expression, assignment-equal instead of test-equal, missing pieces of a compound logical expression and incorrect operator for compound statements (AND instead of OR).

MANIFESTATIONS: These errors manifest themselves in various forms: incorrect result, failure in particular runs, program not behaving as expected, unexpected and possibly unrelated exceptions such as null pointer exceptions.

DIFFICULTY AND FREQUENCY: The defects behind these errors are moderately difficult to locate since these may have cause and effect separated in space and time (cause/effect chasm [16]), attract faulty assumptions or mis-directed blame and are generally buried in unstructured (spaghetti [16]) code. Frequency of occurrence is high, since this error occurs in structures that form the basic building blocks of the code.

PLUG-IN DESIGN: These errors require a very low level visualization, one that can display values of variables involved and is capable of single-stepping through a particular piece of execution. A well designed plug-in would present an abstract model of the underlying functionality so that the user can detect an anomaly without being overwhelmed by the details, such as tracking a few variables of interest, a few methods or all methods of a particular class. A technique that allows the users to visualize values of variables just before and just after the suspected bad code would be very useful in tracking down the fault.

PLUG-IN INPUT REQUIREMENTS: Input requirements would include tracking particular variables (events on initialization, modification and termination) apart from possibly augmenting this information with line numbers of the associated code. It would be required to control the execution (for single stepping) for generating events as each line is executed informing about the current values and system state. Some knowledge of the code may prove useful to establish the context of the variables (i.e. whether they appear in a while loop, or they are class members and would persist, or they are variables local to a method)

POSSIBLE DISPLAY: The plug-in may look as shown in Figure 5.2. The figure shows variables of interest being tracked down beginning from the initialization stage to their termination, depicting when (in time) and where (in code) were the changes made, and the resulting new value. Time is displayed on the horizontal axis and the parameter value on the vertical axis. Three independent integer variables are being visualized, *i*, *j* and *num*. Initialization of a variable is marked by the keyword “init” and termination (going out of scope) by keyword “term”. The plot may help infer some general information about the variables, such as the monotonically increasing behavior of *j* implies that it is most probably involved in some kind of iterative construct.

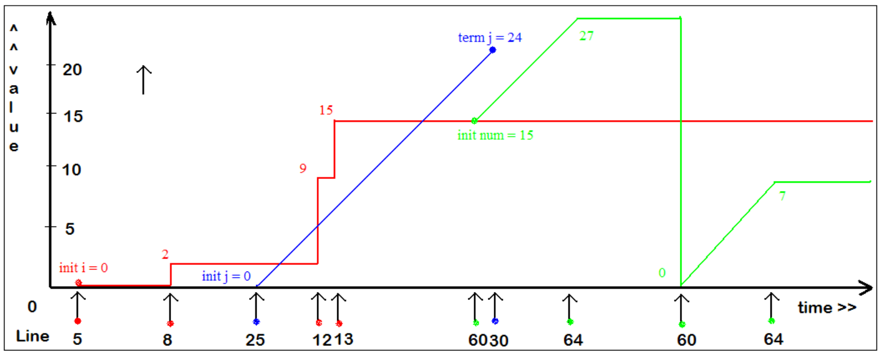


Figure 5.2: An example visualization for detecting incorrect conditions for logical expressions.

An advanced feature displayed in the figure involves relating the modification of parameters of interest with the line of code where the modification took place. The figure shows the associated line of code by adding a second index (“Line”) to the horizontal axis, in addition to “time”. The upward pointing arrow below the horizontal axis followed by a number signifies the line of code where the change took place. The arrow has a colored dot as its tail for easy recognition of the variable it is associated with (the color of the dot is the same as the color used to plot the variable). So a glance of the figure provides a lot of useful information, for example, variable *i* was declared and initialized to 0 on line 5, and assigned the values 2, 9 and 15 on lines 8, 12 and 13 respectively. In general, many variables can be tracked at the same time, although the display may be used judiciously by depicting only a selected set of parameters at a time. Such graphs can be generated asynchronously (meaning between any two defined points (visual breakpoints) in the code) and the behavior during those particular phases can be studied.

The behavior shown by variables must match the programmer’s intuition. Anomalies such as incorrect loop iteration will have an incorrect final (or initial) index value. Nuances such as incorrect set-up may be detected wherein a loop which should have executed 25 times (i=1 to i=25) did actually execute 25 times, although the indices went from 0 to 24. Incorrect values assigned, values changed by objects that were not supposed to change them and other similar errors can all be caught using the graph.

Incorrect Switch Case Statements

Switch-case statements are convenient constructs to route the execution when there are multiple options. However switch-case constructs present problems of their own, and have been identified in [38]. Like other logic errors, these errors may result in control flow errors.

EXAMPLES: Missing/wrong default case, missing cases, fall over (missing break statements).

MANIFESTATIONS: Incorrect result on specific configurations, possibly unrelated run-time exceptions, abnormal program behavior.

DIFFICULTY AND FREQUENCY: Defects causing switch case errors may be difficult to find due to cause and effect separated in space and time (cause/effect chasm) and unstructured code. The frequency of occurrence is very small since the scope is confined to switch-case constructs.

PLUG-IN DESIGN: An effective plug-in would be able to trace the sequence of execution at the level of the method body, and present the information by some intuitive means, for example, by coloring the lines of code executed. Any abstract model of the same, such as one that displays the “case” number executed for every switch construct, would be helpful. Actually real-time analysis may not be required; the color coded code might be enough to spot the error.

PLUG-IN INPUT REQUIREMENTS: First and foremost, the plug-in must have a capability of detecting switch constructs, and enabling / disabling / listening to events after the control enters the construct. Also needed is a way for the users to specify exactly which switch they are looking to visualize, and to uniquely identify the same amongst all others within the code. Each execution must be tracked; an association with line of code may be very beneficial in pin-pointing the error. Some structural information informing the various cases and case boundaries would be required; a very detailed and localized picture is the requirement for detecting these errors.

POSSIBLE DISPLAY: A possible display for detecting these errors is shown in Figure 5.3. The switch construct is depicted by a blue rectangle, with an entry point (on the top left) and an exit point (bottom right) of the rectangle. A statement in red just above the box provides information about the current control parameter (num) and the location of the switch construct in the code, such as the line number, method name and class name. The cases within the construct are depicted by blue horizontal bars, except the *default* case that is represented by a red bar. All the cases with a break statement are identified by placing a red dot at the end of the bar. Finally, four control flow situations are represented by colored lines entering the construct, executing the required case(s) and then exiting the construct.

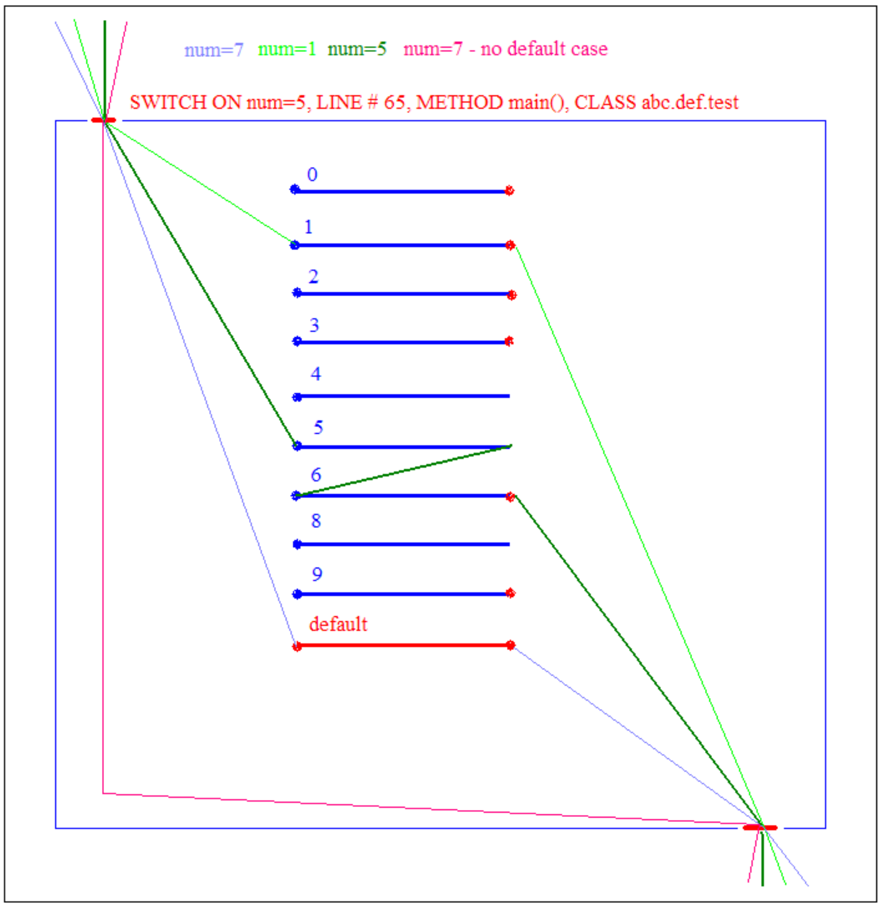

num=7 num=1 num=5 num=7 - no default case
SWITCH ON num=5, LINE # 65, METHOD main(), CLASS abc.def.test
0
1
2
3
4
5
6
8
9
default

Figure 5.3: An example visualization for detecting incorrect switch case statements.

As can be seen, in a situation where the control parameter num = 5 (control flow depicted in dark green), cases 5 as well as 6 are executed (due to the absence of break at the end of case 5). A case where num = 7 (control flow depicted in light blue), the default case was executed (since case 7 is missing). Considering the control flow in pink that assumes that the *default* case was not at all specified, the situation where num = 7 simply passes through the construct without executing any case.

This approach assumes that a lot of information is known, such as all the cases in the switch construct, cases with and without break statements, and whether or not a default is present. A stripped down version without this information can also be generated and it would be reasonably effective in detecting the switch construct errors.

Any anomaly in control flow, such as ones due to missing cases, missing default or missing break statements, would be immediately noticed by aberrant flow of execution within the construct. This cannot assist detection of wrong cases, although such errors may be detected due to a disparity between the expected and visualized behaviors.

Prefix vs. Postfix Operators

Many modern languages such as Java and C++ provide a shorthand method of incrementing/decrementing integer variables: prefix and postfix operators. Using one in place of the other may result in boundary errors in case of loops (loop executing one too much or one too few) or control flow errors in case of conditional constructs. This error category has been identified in [38].

EXAMPLES: Interchanging prefix and postfix operators or using one in place of other.

MANIFESTATIONS: Incorrect result, exceptions such as array index out of bounds.

DIFFICULTY AND FREQUENCY: Defects behind these errors may lay hidden in an unstructured piece of code. These defects are not very difficult to spot since the defect is not too separate from the fault (small cause-effect chasm) although deceptively similar prefix and postfix operators may confuse the user and cause him/her to assign faulty blame to other pieces of code. Frequency of occurrence is very small due to localized scope of the defect.

PLUG-IN DESIGN: An effective plug-in would be able to depict the variables and its state as it changes, especially if coupled with a single-stepping kind of execution. The state events may be initialization, change, and going out of scope. As long as this information is provided to the user in the context of the execution, the error can be detected readily. Basically the construction of the plug-in is similar to the one described in Section 5.2.1.

PLUG-IN INPUT REQUIREMENTS: Same as described in Section 5.2.1.

POSSIBLE DISPLAY: Same as described in Section 5.2.1.

5.2.2 Control Flow Errors

Control flow errors are a subset of logic errors, described in Section 5.2.1, although this categorization encompasses many types of errors, described in the sections to follow. A program's behavior is described by the control flow and a control flow error occurs when, in the words of [28], the program does the "wrong thing next". Under extreme cases, control flow errors may halt the program or cause it to produce unexpected results [28].

Loop iterated wrong number of times

Loop iterations are controlled by a control clause, a logical condition. Incorrect loop controlling clauses, due to a logic error in the loop controlling conditional statement, may cause an "incorrect execution sequence" by allowing the loop to be executed an incorrect number of times [3].

EXAMPLES: These errors may be caused due to any of the following:

- Incorrect initialization of loop control variable or incorrect terminating condition,
- Wrong logical construct for loop control, including off-by-one error,
- Wrong inequalities (e.g. $>$ instead of $>=$),

- Testing floating point values for equality,
- Testing conditions that can never be achieved,
- Assignment-equal instead of test-equal,
- Missing pieces of compound logical expressions,
- Incorrect operator for compound statements (such as AND instead of OR),
- Accidentally modifying the loop index within the loop (decrementing / incrementing twice, assigning some value),
- Incorrect logic within the loop (incorrect "loop fall-through" due to break, continue or return),
- Failure to set/reset flag, and
- Improper loop nesting

MANIFESTATIONS: Incorrect results on all or specific runs, exceptions such as array index out of bounds, program not doing what is supposed to do, program halting.

DIFFICULTY AND FREQUENCY: Defects causing these errors are moderately difficult to locate, generally the defect and the error are in close proximity of each other. These errors are very frequent since this class covers errors that occur in building blocks of the code, and also due to the fact that the scope of this class is very wide.

PLUG-IN DESIGN: An effective plug-in would be able to provide information about the number of iterations for any loop, along with the initial, incremental and final state of the loop control variable. The abstraction, for example, may have bars or dots representing each execution of the loop. Infinite loops may be detected by noting that the execution continues indefinitely (practically beyond some "threshold" number of times). An added functionality may be provided by allowing a single-stepping facility for the desired iteration of the loop to capture failures that occur repeatedly on specific iterations.

PLUG-IN INPUT REQUIREMENTS: The first requirement would be to detect a loop construct and then extract the information associated with it, such as the loop control variable, the initial value and the proposed end condition. The events need to be updated at least once each iteration for an accurate portrayal of the execution. The events need to carry information such as the value of the loop index (or loop control variable). Also, similar to the switch case scenario, a way is needed to determine which loop-construct needs to be visualized and a way for the user to identify it uniquely.

POSSIBLE DISPLAY: A possible display is shown in Figure 5.4. The status window shows current status of the loop, the control variable and the control condition. This information may be updated once each loop or more frequently (although once a loop should be enough). A blue box with a circular arrow identifies the loop construct. There is one entry point (on the top left) and more than one exit point, to depict exit due to various reasons such as loop condition evaluating to a false, break statement or exit condition (a return or system exit). Two control flow situations have been identified with light and dark green lines entering and leaving the box. Information such as the type of loop, the loop control condition and initial value of the control variable are provided (in blue) before the control enters the loop; the exit condition is provided after the control exits the loop and is colored depending on the condition. A status window (available only if the loop is run in single-stepping mode) provides information about each iteration of the loop, such as the current value of the loop control variable, its comparison to the last value, the current control condition and possibility of another iteration.

The figure displays two execution situations. Both enter the construct with the control variable initialized to 0. In one case (depicted by dark green), the control exits the loop due to control condition failing, as informed by the exit status. The other execution trace

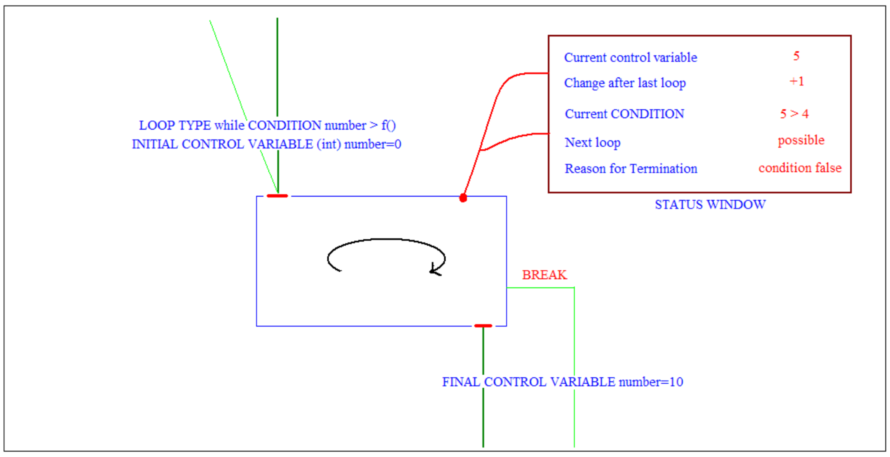


Figure 5.4: An example visualization for detecting loop iteration errors.

(depicted by light green) encounters a break clause that causes it to exit the loop.

The loop control variable is expected to be under close observation and any deviation in its expected behavior will be noticed immediately, either by the value itself or by the change from the last loop iteration. Also, the condition of loop exit is also caught, hence an unexpected break statement ending the loop will be detected. The initial and final values of the control variable will also help detect any problems.

Incorrect flow within If and Switch statements

These errors are exactly the ones categorized under “Incorrect conditions for logical expressions” in Section 5.2.1. The incorrect flow result is due to the wrong set of statements being executed, which is in turn due to incorrect decisions within the if-else clause.

EXAMPLES: The examples include, apart from the ones described in Section 5.2.1, missing sub-conditions, incorrect grouping/association of if and else clauses and incorrect switch case statements (missing/wrong default, missing cases, missing break statements) [38].

MANIFESTATIONS: Incorrect results on specific runs, possibly unrelated run-time exceptions, breaking of some perfectly sound functionality.

DIFFICULTY AND FREQUENCY: Defects behind these errors are generally not difficult to locate. The defect in the code generally manifests itself quickly, allowing fast detection; detecting these is similar to detecting "Incorrect conditions for logical expressions". The frequency of occurrence is low due to the localized scope of these defects.

PLUG-IN DESIGN: Same as that for type "Incorrect conditions for logical expressions" described in Section 5.2.1.

PLUG-IN INPUT REQUIREMENTS: Same as described in Section 5.2.1.

POSSIBLE DISPLAY: Same as described in Section 5.2.1.

Incorrect (sequence of) Method Calls

An incorrect flow of control may result not only due to a wrong decision being taken, but also due to explicit mistakes made while invocation of the methods. The examples below list a few errors of this type.

EXAMPLES: Out of sequence method calls (calling get() before set()), call to wrong method (library function) possibly due to overloading or deceptively similar names.

MANIFESTATIONS: Incorrect result, program not doing what it is supposed to do, exceptions such as null pointer assignment.

DIFFICULTY AND FREQUENCY: Defects behind these errors may cause some confusion due to faulty assumptions or misdirected blame, although these are generally easy to spot. Also, frequency of occurrence is very low since most of these defects are caught by the compiler itself. In case of overloaded methods, faults observed at run-time may be a helpful indication towards presence of these defects.

PLUG-IN DESIGN: A simple method-entry and exit event capturing plug-in may be sufficient to detect these errors. An abstract model may present a UML kind of graph with dynamic method calls depicted by arrows, although the method “objects” need to contain enough information to distinguish among the overloaded methods (such as arguments and return type). An added feature would be to depict the actual values being passed to the method on the arrows signifying the call.

Tracer is a very good example of a plug-in attempting to capture incorrect sequence of method calls. One of the two new plug-ins, Tracer maintains a list of the method names sequenced by their invocation. This footprint of a sample of execution reveals the exact path followed by the program. Tracer has been described in detail in Section 6.1.

PLUG-IN INPUT REQUIREMENTS: Events at the beginning and end of each method call, augmented by the parameters being passed and the return values if any. Tracer relies only on the information about the invocation sequence, it does not deal with method parameters or any other information such as timing relationships.

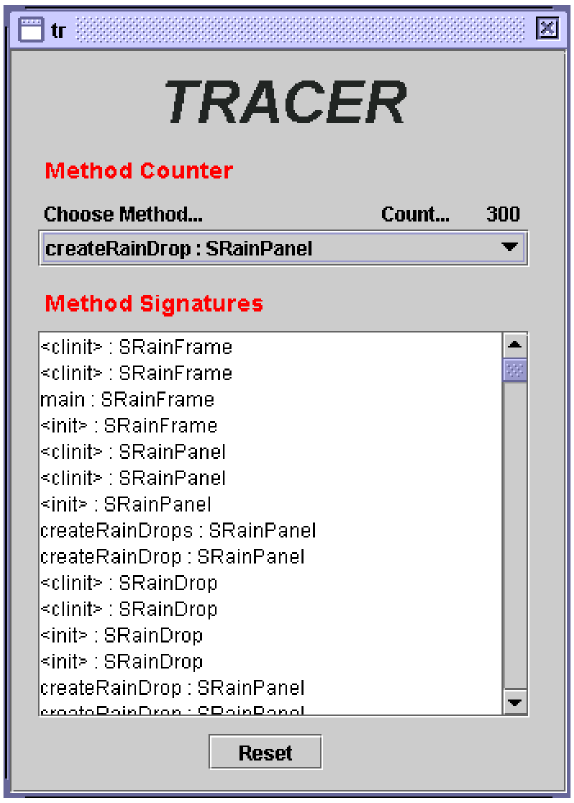
POSSIBLE DISPLAY: Various simple kinds of display are possible. As pointed out above, a dynamic UML look-alike may be good, although problems regarding scaling and screen clutter would need to be taken care of. A flowchart based display is also possible.

Tracer displays the sequence of method invocations. Figure 5.5 shows a screenshot of the Tracer plug-in. Any misplaced or out of order method call would be visible in the display and it is up to the user to have enough knowledge of the software to detect that the calls are to the wrong method or out of order. The plug-in cannot do anything more than display the execution trace.

5.2.3 Computation Errors

Computation errors are the errors in the expressions and assignment statements used to compute output values [22, 3, 18]. According to [28], computation errors may result due to bad logic, bad arithmetic or imprecise calculations.

EXAMPLES: Bad logic includes typing errors (writing A/B meaning to derive a dou-



createRainDrop : SRainPanel

```
<clinit> : SRainFrame
<clinit> : SRainFrame
main : SRainFrame
<init> : SRainFrame
<clinit> : SRainPanel
<clinit> : SRainPanel
<init> : SRainPanel
createRainDrops : SRainPanel
createRainDrop : SRainPanel
<clinit> : SRainDrop
<clinit> : SRainDrop
<init> : SRainDrop
<init> : SRainDrop
createRainDrop : SRainPanel
```

Figure 5.5: Tracer: Visualizing the execution footprint

ble when both A and B are integers), incorrect simplification of a complex expression, using incorrect formulae (or ones inapplicable to current data) and the like. Bad arithmetic errors result due to improper implementation of computing functions. Finally imprecise calculation type errors include round-off and truncation errors. [28]. From a broader perspective, following are examples of computation errors:

- Incorrect operand or operator in an expression.
- Incorrect grouping of sub-expressions (parentheses errors).
- Sign convention error.
- Units or data conversion error.

- Out-of-normal-range values.
- Rounding or truncation errors.
- Incorrect mathematical equations/formulae.
- Unexpected outcome of expressions (peculiarities of integer arithmetic, conversion conventions).

MANIFESTATIONS: Incorrect result, possibly unrelated run-time exceptions.

DIFFICULTY AND FREQUENCY: Defects causing these errors are moderately difficult to locate, however round-off and precision errors may be difficult to spot due to the error manifesting itself much later, both in space and time. Frequency of these defects varies depending on the application (computation intensive or otherwise), although these defects are generally less frequent.

PLUG-IN DESIGN: An effective plug-in would be a low-level data visualization module that can visualize the values of variables the user is interested in, as they undergo computations. The variable may be represented by a colored dot (if it is Boolean) or as a tower of varying height (if numeral) or displayed as it is (if string). The variable(s) should be user-specified and the tracking may be done in real-time (single-stepping mode) so as to pin-point the location of the fault. The design aligns closely with the design of the plug-in discussed in Section 5.2.1.

PLUG-IN INPUT REQUIREMENTS: Same as described in Section 5.2.1.

POSSIBLE DISPLAY: Same as described in Section 5.2.1.

5.2.4 Data Reference Errors

Data errors occur due to incorrect use of a data structures [3]. According to [28], data may be misinterpreted or corrupted while being transferred between methods, modules or programs. This error category have been identified by [22], [3], and [28].

EXAMPLES: Examples of data errors are the use of incorrect array indexing expressions (“off-by-one” indexing error, missing level in indexing) [3], use of a wrong variable in an equation [3], errors in the properties of the data such as dimensions, units [22], and incorrect (or confusingly similar) variable names [16].

MANIFESTATIONS: Incorrect result, exceptions, program not doing what is expected.

DIFFICULTY AND FREQUENCY: Defects causing data reference errors are generally difficult to locate primarily due to faulty assumptions or misdirected blame. Sometimes more problems are encountered in locating these defects due to cause and effect separated in space and time (cause/effect chasm) and unstructured, uncommented code. The frequency of such defects is relatively low.

PLUG-IN DESIGN: A low level data-visualization plug-in would be effective, that can display the values of the corresponding variables and also trace the execution of selected portions of code. The design is very similar to those discussed in sections 5.2.1 and 5.2.3.

PLUG-IN INPUT REQUIREMENTS: Same as described in Section 5.2.1.

POSSIBLE DISPLAY: Same as described in Section 5.2.1.

5.2.5 Interface Errors

Interface errors are the errors that occur while control is being passed between methods [22]. [3] defines interface errors as those that are associated with structures existing outside the module’s local environment but which the module used. Interface errors have been identified in [22], [18], and [3].

EXAMPLES: Examples of interface errors include:

- Incorrect functions used or incorrect subroutine called (possibly due to overloading)

- Values of input variables / order of variables altered.
- Passing by reference vs. passing by value [24, 38].
- Failure to clean up data on exception-handling exit.
- Incorrect assumptions about static and dynamic storage of values (i.e., whether local variable values are saved between subroutine calls).

MANIFESTATIONS: Incorrect result, exceptions being thrown, program not doing what is expected.

DIFFICULTY AND FREQUENCY: Defects behind interface errors are difficult to find due to illusions about the code being right resulting in assignment of blame to other parts of code. Cause and effect chasm also plays a role and spaghetti code can hide the fault. The frequency of occurrence is, however, low.

PLUG-IN DESIGN: Design of a plug-in to detect interface errors aligns well with the design discussed in Section 5.2.2, although some data depicting a global view is also required to establish the context so that values of variables and any changes to them may be captured. The method parameters need to be visualized to capture errors wherein parameters passed to the method are accidentally swapped. Low level variable visualization (such as that discussed in Section 5.2.1) may be helpful to capture the passing-by-reference vs. passing-by-value errors as well as those left behind by exception-handling procedures. Design considerations are similar to those discussed in Section 5.2.2, although the plug-in needs to be more detailed and should be able visualize particular information such as actual parameters.

ParaVis is the first step in this direction. One of the two new visualization plug-ins added to SoftViz, ParaVis depicts the method parameters on a 2 dimensional plot and offers much flexibility in controlling the way the information is presented. ParaVis is described in detail in Section 6.2.

PLUG-IN INPUT REQUIREMENTS: Apart from the requirements detailed in Section 5.2.2, the plug-in would require additional information including the actual parameter values passed and the returned values (if any) for each method invocation. Plug-in ParaVis is very basic and deals only with the method parameters, hence it does not require return values or any other information.

POSSIBLE DISPLAY: Generally, the display would be as described in Section 5.2.2, although a screenshot of ParaVis displayed in Figure 5.6 provides a picture that would be helpful in detecting interface errors due to faulty parameters. As can be observed, generally the chosen parameter stays close to the average value (3) however sometimes it dips as low as 0, and the value remains strictly within 0 and 4. A possibly repetitive pattern is also visible. This is a lot of information inferred from one glance of the plot. Any deviations such as change of the average value, any isolated points (outliers), or a change in the pattern point to possible errors.

5.2.6 Initialization Errors

Initialization errors [28, 18, 16, 3] result due to improper or lack of initialization (or re-initialization) of variables and data structures upon a module's entry or exit [3]. Initialization errors do not only show up the first time a particular method is invoked; it may occur at any point of time during the execution. Also, according to [28], re-initialization failures may be path-dependent, since the values may be re-initialized on one path and not on others.

EXAMPLES: Examples of initialization errors include:

- No initialization of data structures such as arrays.
- Initialization at wrong time or with wrong value.
- No re-initialization of variables between function calls.

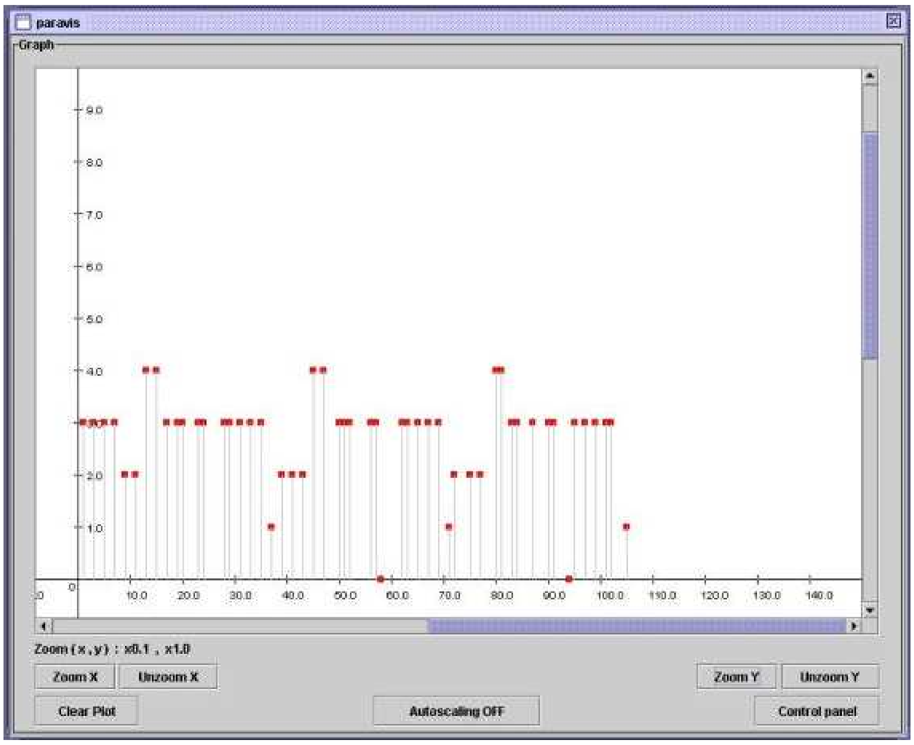


Figure 5.6: ParaVis: Visualizing method parameters

- Wrong or no initialization of loop control variables.

MANIFESTATIONS: Incorrect result, possibly unrelated run-time exceptions.

DIFFICULTY AND FREQUENCY: Defects causing initialization errors may be hard to locate primarily due to faulty assumptions or misdirected blame, apart from the obvious reasons of separation of both space and time between the defect and the fault. These defects are relatively less frequent.

PLUG-IN DESIGN: Design of a plug-in to detect initialization errors is based on the same principles as brought forward in the Section 5.2.3. Further, techniques mentioned for detecting prefix vs. postfix errors in Section 5.2.1 are applicable to initialization errors as well.

PLUG-IN INPUT REQUIREMENTS: Same as described in Sections 5.2.1 and 5.2.3.

POSSIBLE DISPLAY: Same as described in Sections 5.2.1 and 5.2.3.

5.2.7 Boundary-Related Errors

A boundary determines a point across which the situations differ; the program may work in a particular fashion on one side of the boundary while a totally different behavior may be expected or experienced on the other side of the boundary. According to [28] “boundaries describe a way of thinking about a program and its behavior around its limits.” Further, it lists many “types” of limits such as size (largest, smallest), time (latest, oldest, longest) and quantity (most, least) amongst others. Kaner et. al. [28] define three standard boundary errors: mishandling of boundary case, wrong boundary and mishandling of cases outside the boundary.

EXAMPLES: As per [28], examples of “mishandling of the boundary case” include incorrectly handling any possible boundary cases, such as handling $x > 0$ and $x < 0$ fine, but not handling x = 0 gracefully. An incorrect boundary set by the program ($x > 10$ instead of $x > 0$) is an example of “wrong boundary” error. Simply ignoring values on one side of the boundary is an example of “mishandling of cases outside the boundary”, typically due to assuming certain unlikely and unwanted values will never be encountered. Boundary errors may be categorized in the following three broad categories:

- Numeric boundaries
- Boundaries as set by the logic of the program
- Equality of float values

MANIFESTATIONS: Incorrect result, possibly unrelated run-time exceptions.

DIFFICULTY AND FREQUENCY: Defects causing these errors may not be visible in most runs and only some particular conditions cause them to surface. It therefore becomes very difficult to track down these defects in large programs since they may be difficult to reproduce. Frequency of these defects is moderate.

PLUG-IN DESIGN: Design on the same line as described in Sections 5.2.1 and 5.2.3.

PLUG-IN INPUT REQUIREMENTS: Same as described in Sections 5.2.1 and 5.2.3.

POSSIBLE DISPLAY: Same as described in Sections 5.2.1 and 5.2.3.

5.2.8 Concurrent Access and Race Conditions

Concurrent access and race conditions are problems that are faced in multi-threaded programs, where two or more threads compete for resources and end up accessing stale data, uninitialized data, or data that has not been correctly re-initialized, resulting in errors [24]. Deadlocks are a case where a solution to this problem is either incorrectly designed or incorrectly implemented. These errors are one of the toughest errors to capture since they are "irreproducible", they do not appear repeatedly but once in a while. This irregular nature causes problems in trying to locate and correct the defect [28].

EXAMPLES: Examples of these errors include:

- Races in updating data.
- Assumption that one event or task has finished before another begins.
- Assumption that input won't occur during a brief processing interval.
- Resource races: the resource has just become unavailable.
- Assumption that a person, device, or process will respond quickly.
- Task starts before its prerequisites are met.

MANIFESTATIONS: Incorrect results, run-time exceptions, program not responding.

DIFFICULTY AND FREQUENCY: Defects causing these errors are very difficult to locate and correct primarily due to their "irreproducible" nature along with other causes including faulty assumptions or misdirected blame, difficulty in visualizing, illusion that the code is fine, and cause and effect being totally separate in space and time. Frequency of occurrence depends on the type of application (multi-threaded or not) although most of the applications today use the multi-threaded paradigm and these errors surface often.

PLUG-IN DESIGN: An effective plug-in would capture various threads and their states (executing, blocked or waiting on a resource) and their interaction, if any. A global view must be maintained at all times with focus on threads when something special happens, such as a thread accessing objects/variables accessible by other threads. Visualization may be on the lines of UML diagrams showing different sequences of execution by different lines each having their own set of events. Deadlocks can be detected by noting entry into a method but no exit for a specified threshold of time.

PLUG-IN INPUT REQUIREMENTS: Information about various threads of execution would be required. The current state of each thread (such as suspended, executing or blocked) would also be useful information.

POSSIBLE DISPLAY: Figure 5.7 shows a possible display. Different threads are depicted in parallel along the vertical axis, with time as the independent parameter displayed on horizontal axis. The creation of each thread is captured, along with information about the parent creating the thread. Other states such as executing or blocked are captured, and the associated methods are noted (in which the thread started execution or is currently blocked). The state of execution is depicted by green and a blocked state is represented in red. Other metadata such as the message passing between threads, the location in code where a thread gets blocked and the cause of it getting unblocked may also be captured for greater understanding.

Some error conditions, such as an incorrectly blocked thread (any thread always blocked and hence blocking everyone at the end) may be readily caught. Also, performance bottlenecks may be observed by noting the threads' states. However, there are various errors that would not be caught by this model. Note that Figure 5.7 is based on

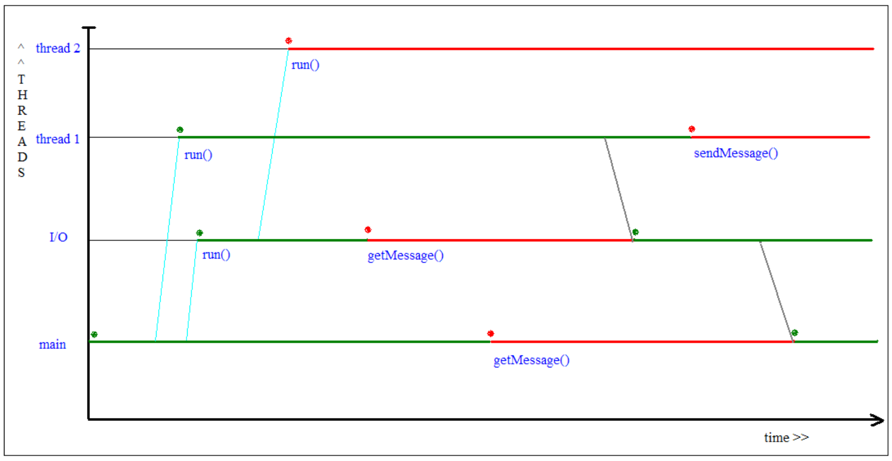


Figure 5.7: A visualization dedicated to capturing race conditions and thread synchronization issues.

our opinion and experiences, and may not be accurate or even complete. More powerful abstraction techniques and more detailed information are required to capture these errors.

5.3 Framework Summary

This chapter presented an error classification framework that categorized common errors into classes and related these error classes with the visualization scenarios that will be helpful in detecting them. Various classes of errors were explored and their runtime manifestations were outlined. Further, relative difficulty of locating the defect behind the error and the frequency of occurrence of the error were estimated for each category, a summary of which appears in the Figure 5.8. This was followed by a detailed description of abstract models that could be developed for detecting errors of a particular class, including their input requirements and possible on-screen display.

Figure 5.8 summarizes the difficulty of locating the defects that cause the errors in each category as well as the frequency of occurrence of the errors. The plot has been divided into three regions:

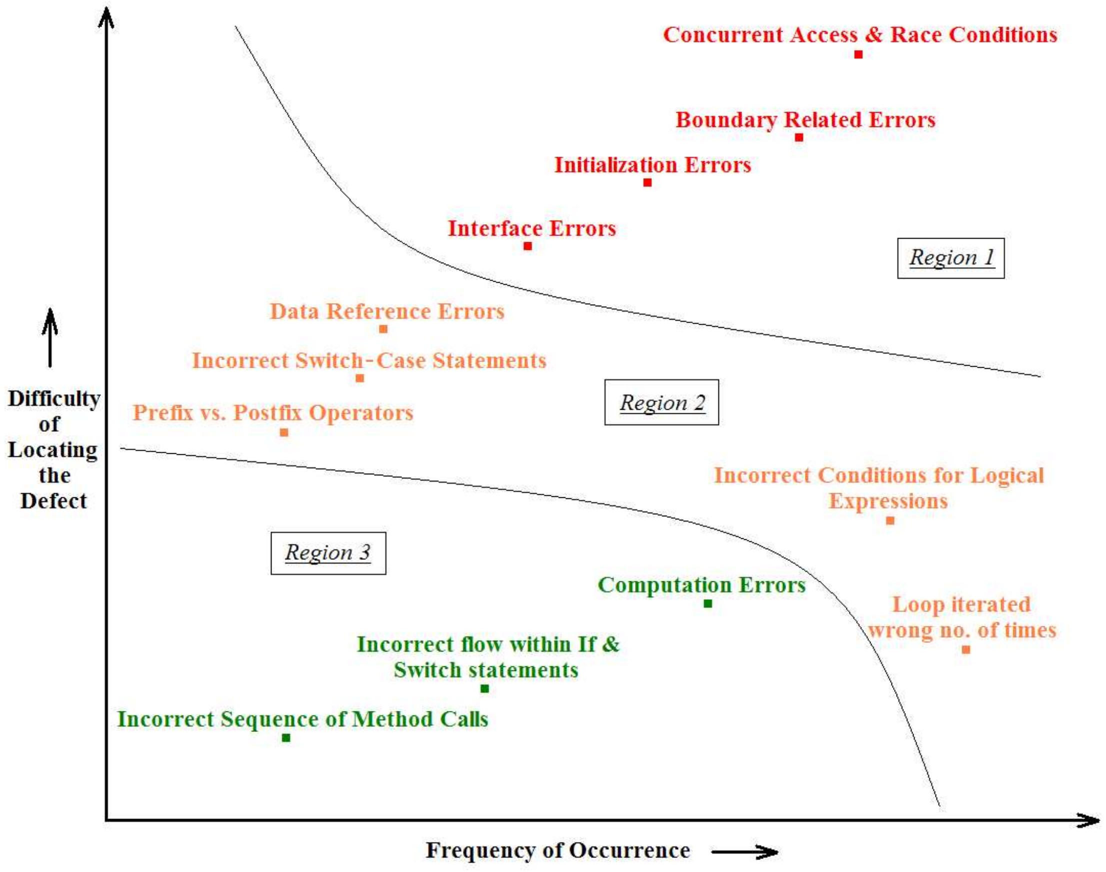


Figure 5.8: A plot summarizing the difficulty of locating the defects and frequency of occurrence of errors.

- Region 1: Error categories in this region are most troublesome since defects causing the errors in these categories are very difficult to locate, and these errors occur more often than others. Visualization plug-ins are required to ease the detection of errors belonging to categories in this region.

- Region 2: Error categories in this region are relatively less frequent, and defects behind these are relatively easier to locate. Plug-ins may be developed to detect errors in these categories, however conventional techniques may be used as well.

- Region 3: Error categories in this region are either very rare, or the defects behind them can be located readily using the conventional techniques. It is advisable to use the conventional techniques to detect errors in these categories.

This error classification framework does the necessary groundwork to set the foundation for development of effective and dedicated SoftViz plug-ins. We believe this error classification framework not only benefits SoftViz and software visualization, but is also an important contribution to the field of software engineering.

Chapter 6

New Plug-ins

The plug-in based architecture of SoftViz allows independent development of visualization plug-ins that could later be simply plugged into the system. The original system had two plug-ins - Sunburst and Timeline. This thesis contributes two new plug-ins to the system - Tracer and ParaVis. Tracer presents a list of the method invocations as the target application executes, hence generating a trace of the program. ParaVis is the first step into the world of data visualization; it visualizes the method parameters.

6.1 TRACER

Tracer is a plug-in aimed at detecting control flow errors; it lists all the accesses made during an execution and keeps a count of each unique access.

6.1.1 Principles

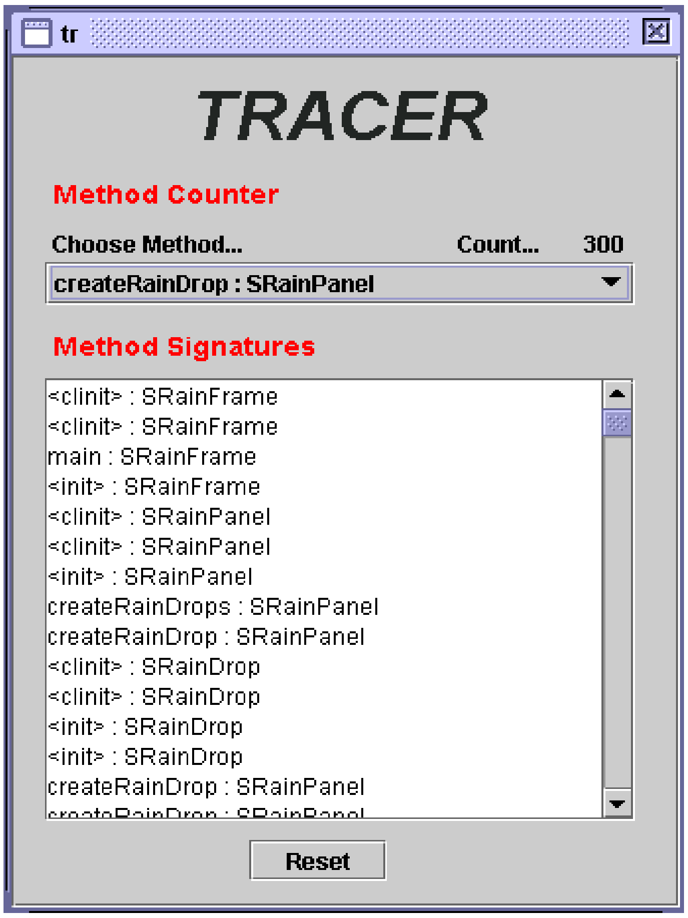
The Tracer module, as its name suggests, traces the flow of execution of the target application. As each event occurs, the associated method name gets displayed in the text area, providing a simple list of method accesses made by the application over time. There is a facility wherein a count of each unique method access is maintained. This visualization option is useful in cases where method calls are made out of order, or when a method gets invoked an incorrect number of times. This module only puts forward the idea wherein a trace of the execution is presented to the user with some metadata attached, which may be helpful in detecting errors in the control flow. More intelligent plug-ins may be developed that present information at a more abstract level, present information filtered based on some user-criteria, or generate more pertinent metadata by internal-processing of the information available.

6.1.2 Implementation

A screenshot of the Tracer visualization appears in the Figure 6.1. The figure shows Tracer tracing the control flow; a list of method : class-name pairs appearing in the window constitute the execution footprint. As can be seen, the user has selected a particular method-class pair from the combo-box (createRainDrop:SRainPanel) and Tracer displays the total number of times the method was invoked (300).

We observe that Tracer does not really "visualize" information in the strict sense; it delivers a simple text-based output which is the list of method invocations. The text-based approach may not agree with the general idea of "visualization", however counter-intuitively, visualization refers to mental images rather than images drawn by the visualization tool. We believe our approach is a simple and effective way to present control flow information, however one can come up with complex abstractions that may use animation to visualize the same data. Moreover, various enhancements are possible in the current implementation, such as showing all the methods and their invocation counts in a list and allowing operations such as sorting on class names, method names or count.

This visualization provides the user with a sense of the order in which the control flows, i.e. the path taken by the execution depicted by methods accessed. Consider a



createRainDrop : SRainPanel

```
<clinit> : SRainFrame
<clinit> : SRainFrame
main : SRainFrame
<init> : SRainFrame
<clinit> : SRainPanel
<clinit> : SRainPanel
<init> : SRainPanel
createRainDrops : SRainPanel
createRainDrop : SRainPanel
<clinit> : SRainDrop
<clinit> : SRainDrop
<init> : SRainDrop
<init> : SRainDrop
createRainDrop : SRainPanel
```

Figure 6.1: The TRACER visualization

possible error situation where a method ”getGrade” appears before any invocation of the method ”SetGrade”. The execution trace produced by Tracer would reveal this anomaly and the error would be detected much before it manifests itself as a fault later in the program.

The information brought forward by Tracer may be helpful for optimization purposes, apart from understanding the program flow, since the method access count reflects the total number of invocations of each method. Assume a method (say the paint method of a GUI component) being invoked 100 times out of total 150 method invocations in the program, it is a good idea to optimize this method to get a noticeable change in performance.

6.2 PARAVIS

The plug-in ParaVis is the first in SoftViz to explore the world of data visualization. This plug-in depicts the method parameters on a 2 dimensional plot and offers much flexibility in controlling the way the the information is presented.

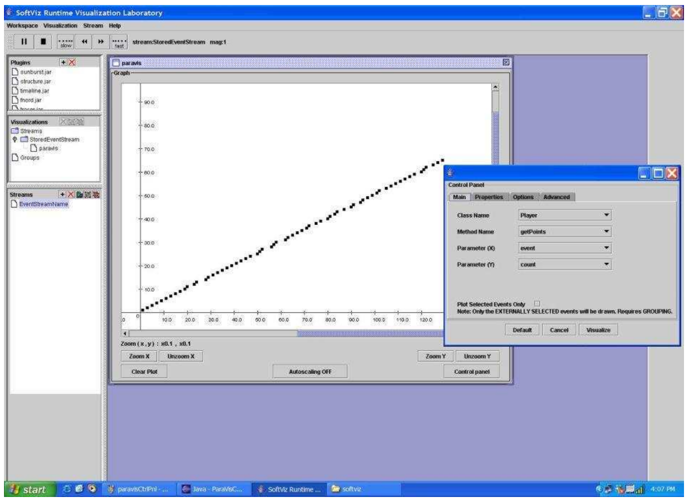
6.2.1 Principles

The ParaVis module conveys information about the relationship amongst various accesses of a particular method. The parameters passed to the method are displayed on the plot providing both a high-level view of the general trend and the low-level information about particular values received in each access. The count of invocations of any method can also be visualized in a time referenced manner. The temporal relationship between parameters from different accesses can be explored by choosing time on the reference axis (axis on which the reference parameter such as time or count is plotted), or the inter-relationship between the parameters may be visualized by choosing parameters on both the axes. Time-independent and event-ordered information about the parameters (or the count) can also be presented.

6.2.2 Implementation

Figure 6.2 provides a screenshot of the ParaVis visualization. The figure shows the plot of the number of accesses of a particular method selected by the user with respect to the total number of events. The ParaVis control panel is also shown, used to select the desired graphing options of ParaVis.

ParaVis is composed of two sections: the graphing section, where the plots appear, and the control panel that controls the look-and-feel of the graph as well as the visualization options. ParaVis is a very flexible visualization module with customizable axes and



Class Name: Player
Method Name: getPoints
Parameter (X): event
Parameter (Y): count

Plot Selected Events Only
Note: Only the EXTERNALLY SELECTED events will be drawn. Requires GROUPING.

Figure 6.2: The PARAVIS visualization.

numerous plotting options. Most of these options are accessible from the ParaVis control panel. The user is provided with a drop-down list of all the classes being visualized. Once the user selects the class, another drop-down list displays all the methods of the chosen class. Once the user chooses the method he/she is interested in, two more drop down-lists present the user with the plotting options on the two axes.

On the plot axis, options include method access count and all the parameters of the chosen method. The user may choose to visualize the count or any particular parameter against the option chosen on the reference axis. On the reference axis, the options are time, event and parameters of the chosen method. Choosing "time" allows the user to visualize time referenced information; choosing "event" simply displays the time-ordered set of values. Choosing any parameter enables visualizing one parameter against another, a mode very useful for GUI applications that take in x and y coordinates for the size of the display or for plotting values on the screen.

Graphing options include the type of graph: point graphs (for low level analysis of parameter values) and line graphs (for a high-level view of the general trend). More options such as bar-graphs (for depicting relationships between parameter values) and piecharts may be added to the list. The user may switch-on drops, which are lines joining the plotted values and the reference axes. The drops may be along either or both axes, and the color of the drop is configurable. Further, a grid may be switched-on in the background for providing better reference for the plotted values. The grid may be along one or both axes, and there are many options for grid color. The grid lines themselves may be dotted, dashed or solid. A screenshot of ParaVis shown in Figure 6.3 displays grid being used. Elision is provided by independent "zoom in" and "zoom out" buttons for each axis, and a "spacing" slider that provides fine control over the scale for any particular zoom level. Hence a lot of freedom is provided for the user to select the most appropriate elision settings to suit the application at hand.

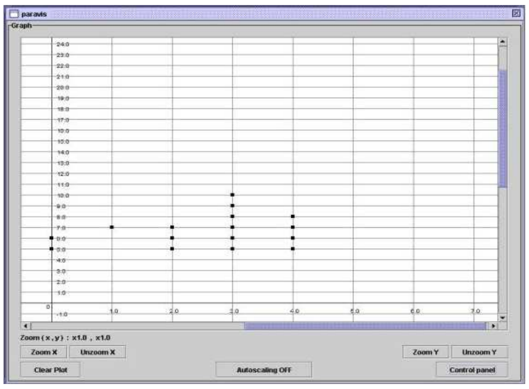


Figure 6.3: ParaVis visualizing one parameter w.r.t other

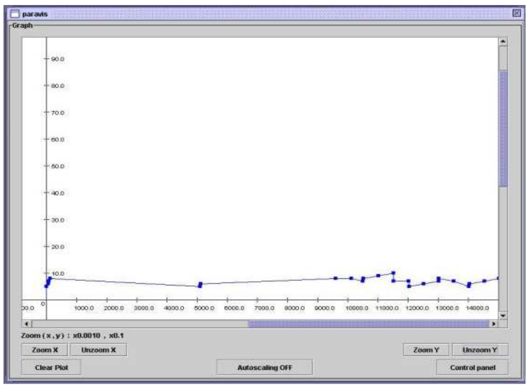


Figure 6.4: ParaVis visualizing method parameter w.r.t. time

This visualization is especially helpful for detecting erroneous data values (data errors), interface errors and boundary conditions. Further, patterns as seen on the graph reveal general behavior of the parameters being passed to methods (such as monotonically increasing, constant over time or repetitive patterns). Any abnormalities in the plot may be a hint of initial manifestations of error and this may help detect errors much before they can assume a form that is difficult to analyze. Figure 6.3 shows ParaVis visualizing one parameter w.r.t. other. The positioning of points is indicative of the relative values of the parameters. Further, the overall pattern of the plot describes the general behavior of the parameters. Such a plot may be helpful in detecting interface errors, especially those caused due to interchanged parameter values. Section 5.2.5 presents another case detailing how ParaVis helps detection of interface errors.

Figure 6.4 displays ParaVis visualizing a parameter w.r.t. time. A glance of the plot reveals the general behavior of invocation of the chosen method at the startup of the application. It can be seen that the behavior is different in the later stages; the number of invocations increase and also the invocations are more spread out. The value of the parameter remains between 5 and 10 for most part. Any change in the general behavior with time (such as the one noted above) and any points on the plot that are isolated from the overall behavior indicate the possibility of an error.

6.3 Plug-in Summary

This chapter discussed the new plug-ins added to SoftViz: Tracer, which traces the execution sequence, and ParaVis, which visualizes the method parameters. SoftViz now contains four visualization plug-ins in its repertoire, including Sunburst and Timeline, inherited from the previous version of SoftViz. Each of these plug-ins can be used to explore various abstract models that aim at simplifying system understanding, however there are numerous other models that need to be explored. The error classification framework proposed in Chapter 5 provides many ideas that can be pursued to develop intelligent and effective visualization modules.

Chapter 7

Conclusion and Future Work

The work of this thesis focused on enhancement of the SoftViz environment. This chapter concludes the thesis and brings forward some of the avenues that exist for future work.

7.1 Conclusions

The thesis has met the main goals set forth as visible from its major contributions as described below:

7.1.1 Revised SoftViz

The first contribution of the thesis is the enhanced SoftViz system. We completed the incomplete work and added new functionality to remove possible bottlenecks as well as to create a platform that may be used for development and testing of efficient visualization plug-ins. Revised SoftViz extracts events directly from the JVM negating the need of instrumenting the source code. Further, SIENA is no longer required for operation which saves a lot of overhead. SoftViz is now a more efficient and versatile software visualization framework.

7.1.2 Integration with Eclipse

One of the most important contributions of the thesis is the integration of SoftViz with Eclipse, enabling possible widespread use and greater convenience especially while development.

7.1.3 Error Classification Framework

Another important contribution of this thesis is the development of the error classification framework that will serve as a guideline for development of more visualization options to further enhance the effectiveness of SoftViz as a debugging tool.

7.1.4 Plug-ins

This thesis designed and developed two new plug-ins for the SoftViz framework, namely Tracer and ParaVis. Another contribution of the thesis is the enhancement of existing plug-ins, Sunburst and Timeline.

7.2 Future Work

The plug-in based architecture of SoftViz allows SoftViz to be extended by development of more visualization plug-ins. The error classification framework proposed by this work lays the foundation for development of dedicated plug-ins that can effectively detect errors belonging to particular error categories.

An interesting area of research is the abstraction models that are easy to understand and visualize. If the abstract model produced by a visualization corresponds closely with the mental model of the user, the user may find it easy to relate with the visualization and hence can more accurately follow the behavior of the program. Thus we need to understand the mental model that humans use for understanding code, as this would help in designing effective visualization plug-ins.

Event extraction remains a heavy process and the less the number of events extracted, the better is the system performance. However, the fewer the number of events, the less is the information that can be visualized, resulting in ineffective visualizations. The JPDA architecture provides opportunities for specifying the particular types of events that need to be extracted. Depending upon the amount of information desired, the event extraction utility can be configured to extract only the required events, resulting in better system performance and effective visualizations. Such a configuration utility would require effective communication between the plug-ins, the SoftViz core and the event extraction utility. Development of this utility would help extract only the required events and hence gain both the advantages as mentioned above.

A facility for coalescing similar events into an aggregate “meta-event” may be developed. This would not only reduce the system load (by reducing the number of events transferred) but may also prove helpful for visualizations that may use the meta-events better than the individual events. For example, a plug-in involved in identifying iterative structures may not be interested in the events that occur in each iteration, hence all the events that correspond to iterations can be coalesced into one meta-event and only this may be transferred to the plug-in.

The JPDA based implementation of SoftViz has a limitation: the system does not have access to actual parameter values passed in method invocations due to the inability to access parameter values in real-time (without suspending the system using breakpoints). This is basically a limitation of the JPDA sub-system and we hope future releases of JPDA will support direct extraction of parameter values. Overcoming this limitation would be a valuable contribution to the system. The SIENA based operation does not suffer from this problem.

Sun Java [55] has announced enhancement of JPDA [27] in the new release of the Java family (version 1.5). Enhancements include addition of a new native programming interface Java Virtual Machine Tool Interface (JVMTI) and addition of new methods to the Java Debug Interface (JDI), among various others. It would be beneficial to perform a detailed study of the new and modified interfaces to assess whether these improvements translate into any advantages to SoftViz.

More attempts must be made at integrating SoftViz at the second level of integration, “invocation”, as defined in section 4.1.2, or even higher. The more seamless the integration, the more convenient would be the process of invoking SoftViz and using it for debugging.

SoftViz currently uses files for buffering of live events and storage of streams for later playback. A better option is to configure SoftViz to use commercial databases such as Oracle or SQL for both storing and buffering events. XML may be used to store the events. This would result in a more structured storage mechanism and optimized system performance.

The work done in this thesis remains to be evaluated more comprehensively. This includes evaluation of the SoftViz framework as well as its plug-ins. This could be accomplished by performing user studies involving developers as well as students; a study may compare the time and effort required to understand/debug a system with and without the aid of SoftViz. The proposed error classification framework needs to be evaluated as well; the error categories as well as the difficulty/frequency estimations may be put to test. The proposed plug-ins may be implemented and evaluated to validate the error classification framework.

Bibliography

[1] S. Asija. Visualization of object-oriented design models. Master's thesis, Depaul University, Chicago IL, December 1999.

[2] R. Baecker, C. DiGiano, and A. Marcus. Software visualization for debugging. *Commun. ACM*, 40(4):44–54, 1997.

[3] V. R. Basili and B. T. Perricone. Software errors and complexity: An empirical investigation. *Commun. ACM*, 27(1):42–52, 1984.

[4] J. L. Bentley and B. W. Kernighan. *A system for algorithm animation - Tutorial and user manual*. W. B. Saunders Company, 1990.

[5] Bluej - an interactive java environment (2004). http://www.bluej.org.

[6] M. H. Brown. Exploring algorithms using balsa-ii. *Computer*, 21(5):14–36, 1988.

[7] M. H. Brown. A system for algorithm animation and multi-view editing. In *Proceedings, 1991 IEEE Workshop on Visual Languages*, pages 4–9. IEEE Computer Society Press, 1991.

[8] M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. Technical Report 75, DEC Systems Research Center, 28 1992.

[9] M. H. Brown and R. Sedgewick. Progress report: Brown university instructional computing laboratory. In *Proceedings of the fifteenth SIGCSE technical symposium on Computer science education*, pages 91–101. ACM Press, 1984.

[10] M. H. Brown and R. Sedgewick. A system for algorithm animation. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 177–186. ACM Press, 1984.

[11] B. Bruegge and A. A. Dutoit. *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. Prentice Hall PTR, 1999.

[12] Software visualization, concordia university, (2004). http://www.cs.concordia.ca/˜rilling/html/software-visualization.shtml.

[13] P. Eades and K. Zhang, editors. *Software Visualization*. World Scientific Pub., Singapore, 1996.

[14] D. J. Eck. Introduction to programming using java, version 4.0. Online Textbook, available at http://math.hws.edu/javanotes/. http://www.faqs.org/docs/javap/source/.

[15] Eclipse white paper (2003). http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[16] M. Eisenstadt. Tales of debugging from the front lines. In J. Spohrer and C. Cook, editors, *Empirical Studies of Programmers: Proceedings of the Fifth Workshop*, pages 86–112, Ablex, Norwood, NJ, 1993.

[17] M. Eisenstadt, M. Brayshaw, and J. Paine. *Transparent PROLOG Machine: Visualizing Logic Programs*. Kluwer Academic Publishers, 1991.

[18] A. Endres. An analysis of errors and their causes in system programs. In *Proceedings of the international conference on Reliable software*, pages 327–336, 1975.

[19] L. J. French. An interactive graphical debugging system. In *Proceedings of the June 1970 design automation workshop on Design automation*, pages 271–273. ACM Press, 1970.

[20] Gnu gdb home page, (2004). http://www.gnu.org/software/gdb/gdb.html.

[21] P. Gill. Probing for a continual validation prototype. Master's thesis, Worcester Polytechnic Institute, Computer Science Department, 2001.

[22] W. E. Howden. Validation of scientific programs. *ACM Comput. Surv.*, 14(2):193–227, 1982.

[23] Ibms jikes bytecode toolkit, (2004). http://www.alphaworks.ibm.com/tech/jikesbt.

[24] Java coffee break - top ten errors, (2004). http://www.javacoffeebreak.com/articles/toptenerrors.html.

[25] Jpda home page, (2004). http://java.sun.com/products/jpda.

[26] Ibm corporation, (2002). jinsight. http://www.research.ibm.com/ jinsight/.

[27] Enhancements to jpda in 1.5 family of the java 2 sdk, (2004). http://java.sun.com/j2se/1.5.0/docs/guide/jpda/enhancements.html.

[28] C. Kaner, J. L. Falk, and H. Q. Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., 1999.

[29] A. Kerren and J. T. Stasko. Algorithm animation. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, 2002.

[30] D. Kimelman, B. Rosenburg, and T. Roth. Strata-various: multi-layer visualization of dynamics in software system behavior. In *Proceedings of the conference on Visualization '94*, pages 172–178. IEEE Computer Society Press, 1994.

[31] B. Kurtz. Softviz laboratory for software visualization. Master's thesis, Worcester Polytechnic Institute, Computer Science Department, 2002.

[32] Georgia institute of technology (2003). lens system. http://www.cc.gatech.edu/gvu/softviz/visdebug/visdebug.html.

[33] Levels of integration with eclipse platform, (2004). http://www.eclipse.org/articles/Article-Levels-Of-Integration/LevelsIntegration.html.

[34] H. Lieberman and C. Fry. Zstep 95: A reversible, animated source code stepper. In *Software Visualization: Programming as a Multimedia Experience (J. Stasko, J. Domingue, M. Brown, and B. Price, eds.)*, pages 277–292. MIT Press, Cambridge, MA, 1998.

[35] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 47–56. ACM Press, 2003.

[36] S. Mukherjea and J. T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 456–465. IEEE Computer Society Press, 1993.

[37] S. Mukherjea and J. T. Stasko. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *ACM Trans. Comput.-Hum. Interact.*, 1(3):215–244, 1994.

[38] Open university - java errors, (2004). http://www.open.ac.uk/StudentWeb/m874/!synterr.htm.

[39] Polka 3-d visualization. http://www.cc.gatech.edu/gvu/softviz/3dcv/3dcv.html.

[40] Georgia institute of technology (2002). polka, samba and xtango. http://www.cc.gatech.edu/gvu/softviz/.

[41] Polka visualization. http://www.cc.gatech.edu/gvu/softviz/parviz/polkaanims.html.

[42] B. A. Price, I. S. Small, and R. M. Baecker. A taxonomy of software visualization. In *Proc. 25th Hawaii Int. Conf. System Sciences*, 1992.

[43] B. A. Price, I. S. Small, and R. M. Baecker. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211 – 266, 1993.

[44] Ibm corporation, (2002). program visualization (pv). http://www.research.ibm.com/pv/.

[45] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[46] S. P. Reiss. Visualizing java in action. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 57–66. ACM Press, 2003.

[47] G. Sevitsky, W. D. Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. In *Proceedings of TOOLS Europe 2001 Conference*, 2001.

[48] Siena home page. http://www.cs.colorado.edu/serl/siena.

[49] J. Stasko. Animating algorithms with xtango. *SIGACT News*, 23(2):67–71, 1992.

[50] J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *Proceedings of the IEEE Symposium on Information Vizualization 2000*, pages 57–65. IEEE Computer Society, 2000.

[51] J. T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.

[52] J. T. Stasko. The PARADE environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95-03, Georgia Institute of Technology, Atlanta, GA, 1995.

[53] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[54] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen. Shrimp views: an interactive environment for information visualization and navigation. In *CHI '02 extended abstracts on Human factors in computing systems*, pages 520–521. ACM Press, 2002.

[55] Sun java home page, (2004). http://java.sun.com.

[56] Trace home page, (2004). http://java.sun.com/products/jpda/trace.html.

[57] D. Ungar, H. Lieberman, and C. Fry. Debugging and the experience of immediacy. *Commun. ACM*, 40(4):38–43, 1997.

[58] M. O. Ward. Xmdvtool: integrating multiple methods for visualizing multivariate data. In *Proceedings of the conference on Visualization '94*, pages 326–333. IEEE Computer Society Press, 1994.

[59] Xmdvtool home page. http://davis.wpi.edu/˜xmdv.

[60] Zeus visualization: A comparison of spinning versus blocking. http://www.research.compaq.com/SRC/zeus/spinblock.html.

[61] Zeus visualization: An animation of multi-level adaptive hashing. http://www.research.compaq.com/SRC/zeus/mlahash.html.