Filtering Run-Time Artifacts Using Software Landscapes

by

Arthur Tateishi

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada
M5S 1A4

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Previous work in software visualization has demonstrated the advantages of presenting programs in a graphical manner in both program understanding and debugging. Among the class of *program visualization* systems are those which operate non-intrusively on program sources. One of the major challenges faced by users of such systems is the problem of filtering the enormous amount of run-time information present during the execution of a large system.

In this thesis, I describe an evolution of the Software Landscape. The Software Landscape is a tool for browsing a software system's static structure using boxes together with arrow and containment relations. My *Run-Time Landscape* includes run-time information, utilizes the Landscape paradigm for the elision of unwanted details, and provides a tightly integrated environment for viewing program artifacts from different perspectives. This prototype is useful in debugging, program understanding, and profiling.

Acknowledgements

I am grateful for the kind support of many people. First among those people, I wish to thank my supervisor Professor Richard C. Holt for his patience, guidance, and constructive criticism. I also thank Professor Dave B. Wortman, my second reader, for his helpful suggestions.

The members of the Turing Group are appreciated for their support and involvement in my work. Many thanks to Spiros Mancoridis, Richard Holt, David Penny, Michael Godfrey, Brian Hall, and Gary Farmaner. Special thanks go to Ron Wessels for implementing the compiler-side of the prototype as I needed it. I would also like to thank Blaine Price, Professor Jim Cordy, and Wayne Hayes for their discussions and input on the Run-Time Landscape.

To my friends in DCS, I thank you all for your friendship and support which made obtaining my M.Sc. a positive experience. My parents deserve special thanks for always standing behind me and raising me with a strong belief in higher education. Also, my brothers have been strong role models for me.

Lastly, the generous financial support of the University of Toronto is gratefully awknowledged.

To my parents
Sadao and Kazuko

Contents

List of Figures

Chapter 1

Introduction

In this chapter, the Run-Time Landscape is introduced along with the motivation for its development and the benefits it provides.

1.1 Motivation

The motivation for the Run-Time Landscape is the desire for better tools to understand and manage complex software systems. Some tools exist which provide for the viewing and management of a software system's code and data. However, they are often limited to small examples or provide inadequate mechanisms to clearly view the information in a large system.

To achieve the goal of creating a more powerful tool, the Run-Time Landscape integrates run-time information and high-level structural design views of software systems into a software visualization and debugging facility.

1.2 Why Visualization?

We all use diagrams and pictures regularly to communicate complex or abstract ideas. We do this without hesitation because we understand through personal experience that pictures often convey a concept or complex relationship quicker and more clearly than the written or spoken word. Imagine the design of a skyscraper described entirely in prose without the use of blueprints. The construction crews would spend an inordinate amount of their time reading and navigating pages of text. Similarly, scientists routinely graph experimental data to spot or confirm trends from large sets of data as opposed to reading long lists of numbers.

There are many examples where visualizations help us understand and deal with large complex systems. Thus, it is natural to examine software systems with the hope of using visualization techniques to help us cope with the growing problem of designing, understanding, and maintaining large and complex software systems.

However, it does not suffice to find ways to output all the information about a system. We must also filter and process the information to elide uninteresting details and highlight pertinent information in an easily comprehensible display. Road maps are an example of this. We could use aerial photographs instead of abstract, schematic diagrams for our maps. Photographs display the same layout information in a pictorial manner. However, anyone who has tried to follow an aerial photograph will agree that a nicely designed, colour-coded roadmap is far easier to use. This is a case where displaying all the information in precise detail can cause a perceptual overload of information as opposed to something more structured without extraneous details such as the exact shapes of churches. Another characteristic of road maps is their use of scale. Road maps are available in several levels of scale to satisfy varying needs.

In the next two sections, I will provide an overview of Object Oriented Turing and the Software Landscape. The Run-Time Landscape evolved from the Software Landscape as a tool for debugging, profiling, and understanding Object Oriented Turing programs by exploring the run-time behaviour of the system.

1.3 Object Oriented Turing

Object Oriented Turing (OOT) is the name for both the language and the software development environment [Mancoridis et al., 1993b]. OOT is a language based on Turing [Holt and Cordy, 1988] that supports concurrent processes, classes, inheritance, and polymorphism. OOT provides the usual constructs available in most strongly-typed procedural programming languages including module level scope control. The current environment is a fast pseudo-code compiler/executor combination using a front-end for the X Window System [Mancoridis, 1992, Mancoridis et al., 1993a]. The Software Landscape Viewer is a design and re-engineering tool integrated into the OOT environment.

1.4 The Software Landscape

The Software Landscape Viewer is a graphical tool useful for creating high-level designs called Software Landscapes as well as the reverse engineering of high-level designs from legacy systems. A Software Landscape relates software design entities with source code or interface descriptions in a consistent manner governed by a set of formal rules defining well-formedness [Holt et al., 1992, Mancoridis et al., 1993a, Penny, 1992]. The Software Landscape Viewer displays boxes and arrows to represent the relationships in a design (See Figure 1.1). Containment relations are also used to group program modules together into subsystems which combine with elision modes to allow the selective hiding of subsystem details. Navigation is accomplished by panning and zooming the display or leaping into and out of containment boxes. Some simple queries to show use and inheritance relations are also provided. All of these attributes come together in the Software Landscape Viewer to provide a tightly integrated system for designing and re-engineering software systems [Mancoridis et al., 1993a].

1.5 What is the Run-Time Landscape?

The Run-Time Landscape is an extension of the Software Landscape for debugging and profiling software systems while maintaining the benefits of the Landscape's hierarchical model. The Run-Time Landscape uses OOT and the existing Landscape Viewer as a basis for the prototype implementation.

The goal of the Run-Time Landscape is to manage and present run-time information about a large software system in a manner that is comprehensible to a programmer. This requires that the information be presentable at different levels of detail useful to the programmer.

The next section outlines the information of interest in a run-time system and the following section describes methods to present information without overloading the user.

1.6 Run-Time Information

A program contains a great deal of information that is useful to a programmer. This information can be separated into static information and dynamic information. Static

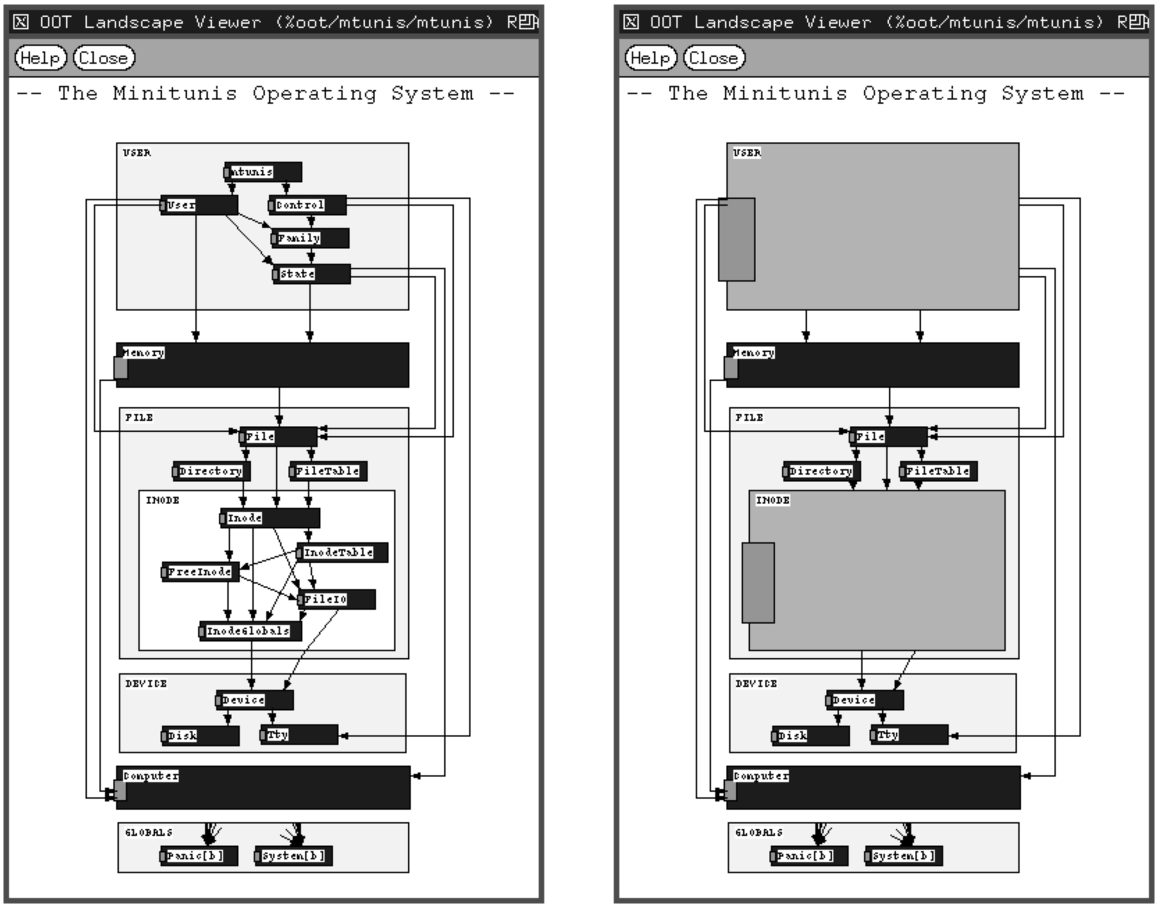

Figure 1.1: **The OOT Landscape Viewer:** This figure shows two snapshots of the OOT Landscape Viewer. The viewer displays a landscape for the Mini Tunis Operating System showing the design level structure of the system. The six top-level stacked boxes are labeled from top to bottom USER, Memory, FILE, DEVICE, Computer, and GLOBALS. The USER subsystem contains five modules and the FILE subsystem contains three modules and the INODE subsystem which contains another five modules. The snapshot on the right has hidden the details of the USER and INODE subsystems. The arrows signify a “uses” relation from one entity to another.

information is that which can be known without executing the program (i.e. by parsing the source code). This would include:

- The identifier names.
- The composite data types and static data structures.
- The subprogram and module structures.
- The caller/callee graph.
- The variables accessible in any scope.

The static information provides a lot of information but it is not complete. Dynamic information is knowledge that can only be obtained by executing the program and observing the state of the program. Dynamic information would include:

- The values of variables and data structures.
- Class instances in an object-oriented system.
- Profiling statistics.
- The execution path.

As stated above, the static information can be obtained by parsing the source code. Sometimes static information is lumped into the dynamic category if is difficult to extract analytically but easy to retrieve during program execution. Blurring the distinction from the other direction, data flow analysis of the code can reveal some dynamic information without executing the program.

Extracting information about a program is one problem in designing any software visualization system. Solving the problem of extracting the information brings forward the even greater challenge of presenting this information to the user in a meaningful way. The next section discusses the steps taken by the Run-Time Landscape to prevent overloading the user with information.

1.7 Presenting Information

Many visualization systems have shown that they can extract a great deal of information about a software system [Myers et al., 1988, Baeza-Yates et al., 1992, Chandhok et al., 1991]. It quickly becomes clear that we must boil off the extraneous information in order to see the interesting events. The number of visualization objects can become overwhelming. Imagine a large OO system with 200 classes. The problem of filtering the information down to a manageable level is formidable but by no means hopeless. Consens and Mendelzon have explored advanced filtering and clustering in Hy+ [Consens and Mendelzon, 1993, Consens, 1994, Mendelzon, 1993].

A good system must be intuitive and easily navigable. Systems which rely on window scrollbars to provide range selection are not very effective. Scrollbar range selectors offer linear panning but they don't provide scaling and the global context is not apparent. Consider our normal use of vision in comparision to wearing a pair of binoculars permanently. The binoculars would offer superior visual detail; however, we normally do not need to examine the dust in the corridor we walk along. We use our eyes to give us a full view of our surroundings and choose to focus our attention on smaller objects only when needed. Figure 1.2 demonstrates how confusing it can be to see too much information at once.

As will be introduced in the next three subsections, the Run-Time Landscape employs three main techniques to provide information about a software system without overwhelming the user. First, the Software Landscape's hierarchical containment views allow the user to choose the desired level of detail in a subsystem down to the subprogram level. Second, the flow of execution is presented using a moving dot called the *execution marble* on the Landscape. Animation techniques are used to make it easier to follow the marble. Last, configurations for displaying variable data are selected in an intuitive manner and the configurations are stored to relieve the user from needing to reselect the same things repeatedly.

1.7.1 Software Landscape hierarchical views

The Software Landscape provides a hierarchical view of a software system. Global context is available and the nested structure allows for fast navigation and the natural elision of details. The Software Landscape approach to filtering information is based on using subsystem

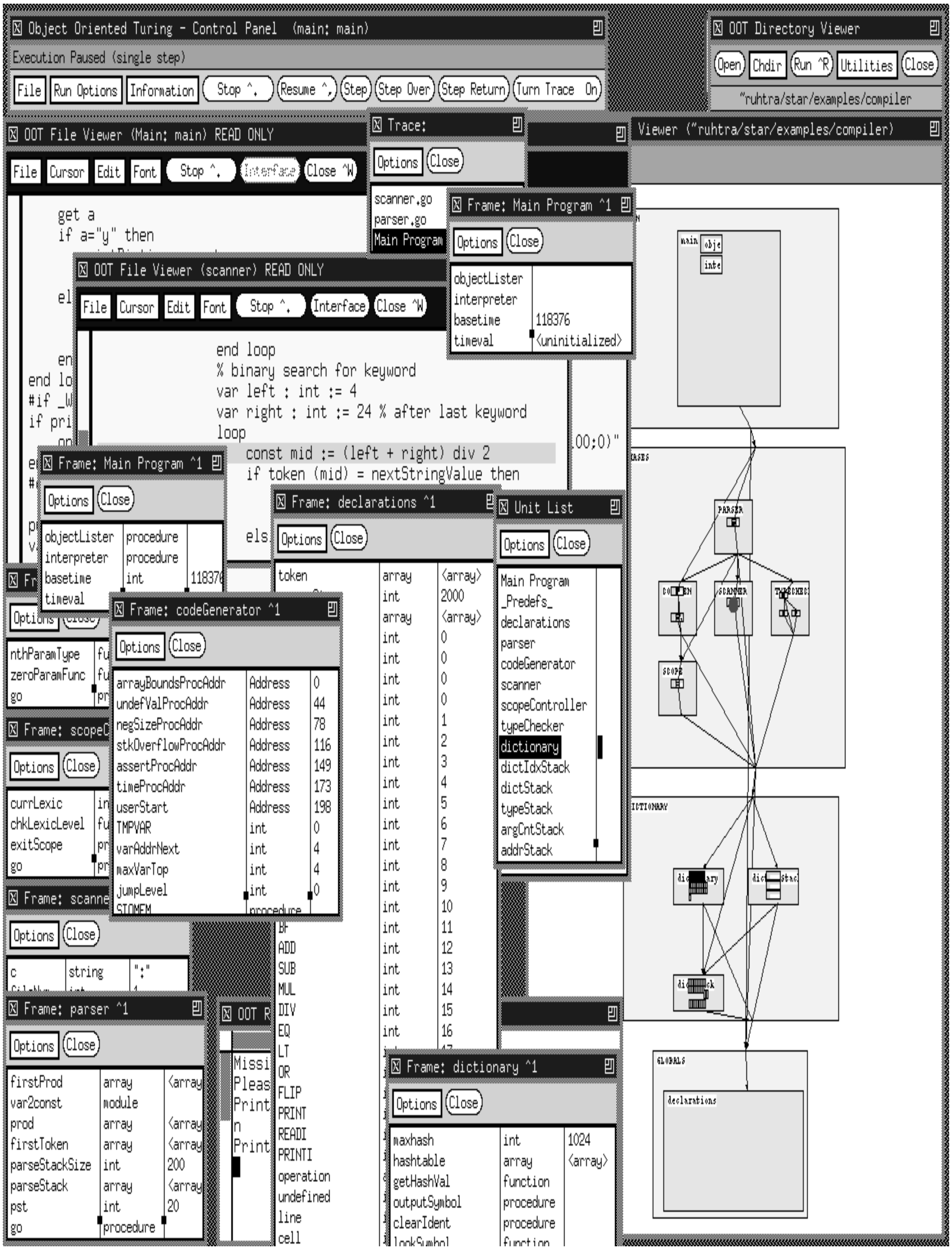

Figure 1.2: **Information Clutter:** In this screen dump of the OOT environment, some of the state of a small compiler is presented along with a landscape diagram. Elision is needed in this picture to make the information comprehensible.

elision to hide substructure details. For example, in Figure 1.1 the user can choose to hide the internal details of subsystems like INODE. The Run-Time Landscape uses this hierarchical structure to simplify navigation of code, to place debugging breakpoints, and to reduce distracting movements of the execution marble. The Landscape also has some options to control the display of arrows.

1.7.2 Animation Techniques

The Run-Time Landscape uses animation techniques to enhance the presentation of the *execution marble*. Subtle cues can significantly help the observer follow an animation. As computers become faster, the display hardware will likely only update at some relatively slow rate (60-80Hz). Therefore, it becomes necessary to give the observer better cues to follow a rapid animation. Cartoon animators have practiced their art for many years. Animators employ anticipation, slow-in/slow-out, motion blurring, arcing, and follow through to present the observer with the illusion of fast motion [Thomas and Johnston, 1981]. Motion blurring is already used by stop motion computer animators in the film industry to create more realistic and convincing pictures of moving objects. By using these techniques in software visualization, the user will be able to follow faster animations with less disorientation [Chang and Ungar, 1993, Hudson and Stasko, 1993].

1.7.3 Selecting Data

Program data is displayed in windows of the *Dataview Tool* and selected using a simple point-and-click interface. The user may also use the Landscape for requesting data to be viewed. All of the Dataview windows can be saved between sessions so the user need not waste time navigating through the same data windows repeatedly. The Dataview Tool and the Run-Time Landscape Viewer complement each to provide a tightly integrated environment allowing the user convenient access to the program's run-time information.

1.8 Thesis Overview

This thesis is composed of five chapters:

Chapter 1: This first chapter introduced the motivation and interest in visualization systems. It described the Run-Time Landscape in general terms of what information it communicates to the user and how this is accomplished.

Chapter 2: In the next chapter, I present some background on the history of Software Visualization and discuss the decisions necessary when designing a visualization system.

Chapter 3: The third chapter will describe the central aspects of the Run-Time Landscape in detail and provide example scenarios for its use.

Chapter 4: This is a technical chapter describing the prototype implementation of the Run-Time Landscape.

Chapter 5: The final chapter concludes with remarks on the value of the Run-Time Landscape to Software Visualization and discusses areas for further research.

Chapter 2

Background

This chapter examines the history of Software Visualization systems, the trade-offs in designing a visualization system, and ends with a summary of where the Run-Time Landscape fits relative to previous systems and what design decisions were made.

2.1 History of Software Visualization

The importance of pictorial representations for computer programs dates back to the use of flow charts to describe control flows [Goldstein and von Neumann, 1947, Haibt, 1959]. As well, most computer science texts make extensive use of abstract diagrams to convey an understanding of data structures and recursion.

The modern renaissance of visualization is widely considered to have begun with the film *Sorting out Sorting*[Baecker and Sherman, 1981] and the availability of high resolution bit-mapped display technology. Some of the successful systems developed include BALSA [Brown and Sedgewick, 1984], a pioneering work in interactive algorithm animation and its successor BALSA-II [Brown, 1988], Incense [Myers et al., 1983] and Amethyst [Myers et al., 1988] which create graphical displays of data automatically, TANGO [Stasko, 1990, Mukherjea and Stasko, 1993] which strives to make smooth algorithm animation easier, and the University of Washington Program Illustrator (UWPI) [Henry et al., 1990] which extracts high-level abstract data structures automatically.

A number of people have written taxonomies categorizing these and other systems. Myers has published a number of revisions of his taxonomy of program visualization and visual programming systems [Myers, 1986, Myers, 1988, Myers, 1990]. An early survey was included in the paper by Glinert and Tanimoto [Glinert and Tanimoto, 1984]. Shu has also published a survey covering a number of systems [Shu, 1988]. Blaine Price, Ronald Baecker, and Ian Small wrote an extension of their earlier work to produce a newer, more comprehensive taxonomy [Price et al., 1992]. The increasing size and complexity of each new or updated taxonomy indicates the rapid growth in the field and the desire to explore new ways to achieve various goals. Unlike the previous taxonomies, Price et al. have developed a taxonomy with a principled derivation [Price et al., 1993]. They define the term Software Visualization (SV) 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.

SV is a more general term encompassing Visual Programming, Algorithm Visualization, and Program Visualization.

The next section defines Visual Programming and the following section defines Program Visualization.

2.2 Visual Programming

2.2.1 Definition

Visual Programming (VP) is the use of graphical methods in the specification of a program. Brad Myers [Myers, 1988] offers the definition:

> Visual Programming refers to any system that allows the user to specify a program in a two (or more) dimensional fashion.

He goes on to argue that conventional textual-based programming is one-dimensional since the compilers that parse such languages do so by treating the program as a one-dimensional input stream. Blaine Price, in his Master's thesis [Price, 1990], criticizes Myers by arguing that programmers may view a textually specified program in a multidimensional manner (lines, and levels of indentation) and it is no concern that the compiler processes the same text as a linear stream. I would counter-argue that inserting blank lines and indentation does not change the program's *specification*, which is a linear stream of tokens, and therefore does not qualify as being Visual Programming.

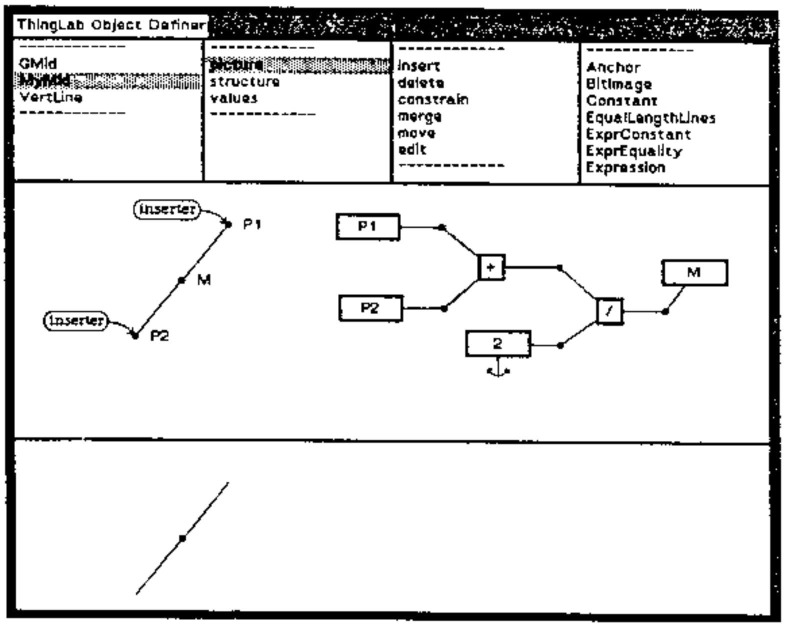

Figure 2.1: **ThingLab:** Screen snapshot of ThingLab showing the definition of MidPointLine graphically.

Given the definition of Software Visualization, the representation of a program in a VP system is a form of SV.

2.2.2 VP Example: Graphical ThingLab

ThingLab [Borning, 1981, Borning, 1986] is a simulation environment that allows users to create and simulate experiments in physics and geometry by specifying the sets of constraints in a system. ThingLab has a graphical interface to program the constraints and data flow of a system (See Figure 2.1).

The next section explains Program Visualization and how it differs from Algorithm Animation.

2.3 Program Visualization

2.3.1 Definition

Simply put, *Program Visualization* (PV) is the use of graphics to enhance peoples' understanding of computer programs.

However, we must be careful to include other media because *visualization* is defined to be the act of forming a mental image. This mental image can be the result of pictures, sound, smells, graphical animations, or any combination thereof [Price, 1992].

Baecker [Baecker, 1986] provided a detailed definition which is clearly a precursor to his definition of Software Visualization:

> Program Visualization is the use of the technology of interactive graphics and the crafts of graphic design, typography, animation, and cinematography to enhance the presentation and understanding of computer programs.

Myers [Myers, 1988] proposed that:

> ... in Program Visualization, the program is specified in a conventional, textual manner, and the graphics is used to illustrate some aspect of the program or its run-time execution.

While I agree that PV entails the illustration of programs or their execution, I see no reason why PV should exclude non-textually specified programs. Such programs will also have control flow patterns and data to represent visually so PV techniques will still apply. Therefore, I prefer Baecker's definition.

Program Visualization addresses the issues of visualizing the code or data of an actual implemented program. This differs from Algorithm Animation where higher-level program descriptions are visualized. The Run-Time Landscape is a PV system by these definitions although I will also use the less precise term Software Visualization in reference to the Run-Time Landscape.

2.3.2 PV Example: BALSA-II

BALSA II [Brown, 1988] is an algorithm animation system that evolved from its predecessor, BALSA [Brown and Sedgewick, 1984]. BALSA II is an effective undergraduate teaching tool that shows a program's high level data structures in a GUI environment. The cost for such a system is high. All algorithms must be manually annotated in the code with calls to indicate interesting events. This annotation phase requires a person familiar with both the algorithm and the BALSA animation system. A labour and skills intensive process such as this can never scale to systems larger than toy examples.

2.3.3 PV Example: Tango and Lens

Tango [Stasko, 1990] is a system for generating algorithm animations similar to BALSA but based on a formal framework. When combined with the Lens [Mukherjea and Stasko, 1993] system, manual annotation is performed with a graphical user interface and code does not need to be intruded upon. Instead, Lens uses the Unix *dbx* debugger to set breakpoints where animation steps are to be performed. The Lens system greatly decreased the amount of work and skills required to produce a useful animation.

2.3.4 PV Example: Visual C

Visual C (VCC) [Baeza-Yates et al., 1992] is a system that allows the end-user to graphically visualize and animate a C program. It functions as a preprocessor that automatically annotates the code with calls to an animation library. VCC's strength is the inherent ability to scale to large systems since it is based on automatic annotation. VCC's weakness is its poor filtering mechanisms which limits usefulness to systems that are manageable on a display screen in their entirety.

A Program Visualization system such as the Run-Time Landscape needs to extract the information and then present the information to the user. These are the topics which will be covered in the following sections.

2.4 Extracting the Information

Visualization systems display information such as run-time objects and the call stack. This information can be obtained by various methods. These methods can be classified by whether they are code intrusive, data intrusive, or use non-invasive probes and whether they are manual or automatic.

2.4.1 Code Intrusive, Data Intrusive, or Probes

In order to extract information about a program, some systems require the source code to be annotated with statements to flag interesting events. These systems are classified as code intrusive (CI) [Henry et al., 1990]. BALSA [Brown and Sedgewick, 1984, Brown, 1988] is an example of such a system. Some code intrusive systems enable the annotation of programs without affecting the original source code. Nevertheless, they still change the executable code generated which may be undesirable.

Data intrusive (DI) [Henry et al., 1990] systems add hookpoints in an object oriented language by subclassing object classes and replacing class methods to suit the needs of the visualization. The Smalltalk animation system developed at Tektronix uses this approach [London and Duisberg, 1985].

Non-invasive probes extract information without affecting the source or object code. The Run-Time Landscape is such a system. The OOT pseudo-code compiler/executor can be polled for information on the state of threads or variable contents. As mentioned earlier, the Lens system [Mukherjea and Stasko, 1993] uses the Unix *dbx* debugger as a non-invasive probe.

The performance penalty for each extraction method depends on the access patterns utilized. The CI method incurs a relatively small penalty per annotation but one pays that penalty during all code execution. The DI method is desirable for data animations or conditional breakpoints since data access events can drive the visualization or conditional checks. However, hookpointing *all* the data in a program would be expensive. Non-invasive probes incur a higher per-use cost than the other methods but don't alter code and don't cost anything when not used. Therefore, intensive probing to monitor changes to a data value can be prohibitively damaging to performance. The three methods are not mutually exclusive so a hybrid approach may be a very desirable option. Data hookpoints could be used to track important data values, code annotations could be used to follow function calls, and non-invasive probes could fill in the gaps by allowing the interactive inspection of data items that were not specifically chosen for hookpointing or deriving the location of execution within a function when needed.

2.4.2 Manual vs Automatic

Visualization systems may rely on the manual annotation of source code or perform the operation automatically. BALSA requires the manual annotation of programs with event statements. VCC [Baeza-Yates et al., 1992] parses C programs and automatically inserts statements to call the VCC visualization support library.

Manual systems have the advantage of having a knowledgable expert (the *animator*) to insert annotations at a high level of abstraction. For example, a series of statements could be identified as a *swap* operation in a sort routine whereas an automatic system will simply note the existence of three assignment instructions. However, this is not always an advantage. Manual systems may require an expert of the software system who is no longer available. Furthermore, a manual system requires a lot time to annotate a piece of source code. Some manual systems such as Lens [Mukherjea and Stasko, 1993] help the animator perform the task of annotation; however, the time to annotate a program is still significant. Therefore, manual systems do not scale up to large software systems whereas automatic systems are far more likely to scale well to large systems.

After obtaining information about a program, we need to process or filter the information before it can be usefully displayed.

2.5 Techniques for Filtering Information

This section discusses the techniques used by several systems to filter information down to a manageable level.

2.5.1 Restricting attention to small-scale systems

The BALSA system [Brown and Sedgewick, 1984] animates algorithms in a clear manner by having an algorithm and animation expert work on small pieces of code, selectively choosing only the interesting events, and providing the most appropriate visualizations for the algorithm. There are two problems with this approach. First, you need to find an expert for both the code and the animation system. Such experts are hard to locate and expensive. Second, the work of annotating source code to label interesting events is very time consuming. The time needed to annotate code makes this approach unworkable for large software systems.

2.5.2 Finding higher level abstractions

Automated program recognition systems parse a program and try to match the code to known clichés to gain an understanding of the program [Letovsky, 1988, Wills, 1990, Lutz, 1992, Wills, 1992]. These systems are still in their infancy and only handle short lengths of code. However, it is conceivable that they will evolve to handle larger systems in the future. Once such a system understands a program, it can in principle decide the best visualization forms to present and assume the role of the expert annotator in a BALSA type system [Price, 1993].

2.5.3 Fisheye views

A fish eye lens shows normal detail at the centre of attention and shrinks and distorts surrounding information to simultaneously show a full view while providing the desired level of detail at the centre of focus. One advantage of these displays is that the user always sees the global context around the focus area so less disorientation occurs [Furnas, 1986, Sarkar and Brown, 1992]. A related approach [Sarkar et al., 1993] uses the analogy of a sheet of rubber being stretched and provides more uniform scaling inside the stretched regions.

2.5.4 Animation principles to increase cognitive bandwidth

The application of animation principles (anticipation, motion blurring, slow in/out, and follow through) in the visualization graphics enables a user to follow and understand faster, more complex visualizations [Chang and Ungar, 1993, Hudson and Stasko, 1993]. Therefore, this approach is useful in almost any system for software visualization. The Run-Time Landscape uses these animation techniques which will be discussed in Section 4.7.

2.5.5 Focusing attention on objects of interest

The GROOVE system [Shilling and Stasko, 1992] has the ability to present an animation and to focus attention on a particular object. In this way, the visualization system filters out information so the user can easily study those interactions which affect the object being studied. For example, the object under focus could be an array of values used in the implementation of a stack. Under such a situation, only program actions which lead to accesses or modifications of the array will be seen in the visualization.

2.5.6 Software Landscape hierarchical views

The Software Landscape provides a hierarchical view of a software system. Global context is available and the nested structure allows for fast navigation and the natural elision of details. The Software Landscape approach to filtering information is based on using subsystem elision to hide substructure details. The Landscape also has some arrow display controlling options. A valid argument against the Software Landscape system is that it only deals with static information. Therefore, it avoids complications due to lots of run-time information.

It should be noted that some of the above techniques are mutually exclusive while others can be combined for a greater net effect. For instance, it is not clear how one would combine fisheye views with Software Landscape views. However, the application of animation principles will improve almost any dynamic visualization.

After obtaining and filtering the information it is necessary to consider when the information will be displayed. This will impact how closely the user can interact with the visualization system.

2.6 Run-Time vs Postmortem Visualizations

There are two principal approaches to visualizing software. One approach visualizes software during program execution (run-time) and the other performs an autopsy on a previously recorded execution trace (postmortem). Both methods have their advantages.

A run-time visualization allows the user to explore a system interactively with the hope that unusual behaviour can be recognized quickly and the relationship between program execution, input choices, and visualizations can be seen directly.

A postmortem visualization has the benefit of complete historical knowledge. The entire execution can be subjected to statistical analysis and the entire time dimension of an execution can be shown at once and navigated at random. Also, full knowledge of the execution enables a postmortem visualization to provide better layouts than a run-time system. A postmortem based debugging and visualization system was investigated using the Hy+ system [Consens and Mendelzon, 1993] for filtering information [Consens et al., 1993].

The Run-Time Landscape follows the example of systems like Amethyst [Myers et al., 1988] and TANGO [Stasko, 1990] in providing visualizations at run-time.

2.7 Summary

This chapter presented background material on Software Visualization and described methods for extracting and displaying visualization information. The key features relevant to the Run-Time Landscape are:

- The Run-Time Landscape is a Program Visualization system.
- Run-Time information is extracted automatically to facilitate scaling to large systems.
- Animation effects and Software Landscape hierarchical views are used to present visualization information.
- Visualizations are displayed in real-time while the program is executing.

The next chapter describes each aspect of the Run-Time Landscape in more detail and discusses its use.

Chapter 3

Run-Time Landscapes

This chapter describes the Run-Time Landscape in detail and how it can be used.

Exploring software architectures using the Software Landscape model is simple and effective. Programmers write software using language constructs like procedures and modules. These are presented in the Software Landscape as naturally hierarchical views. In fact, a programmer's mental model of a software system may closely resemble the form presented by the Software Landscape.

The *Run-Time Landscape* extends the Software Landscape for debugging and profiling while maintaining the hierarchical model. This model is used to interact with a program to reveal desired levels of detail and navigate through the code and data. In effect, the *Run-Time Landscape* becomes *real* to the user because it includes both the dynamic and the static information of a program and hence there is a smaller conceptual gap between the source code and the landscape. It then becomes natural for the user to interact with *Run-Time Landscape* entities instead of the source code for higher-level operations. An integral part of the *Run-Time Landscape* is the use of the Landscape paradigm for the elision of unwanted details.

In the next sections, I will describe the artifacts being conveyed and the tools for presenting the artifacts. The last section of this chapter provides example sessions for debugging, profiling, and program understanding showing how the tools can be used. Most of the examples in these sections are based on a 9000 line compiler written in OOT for a senior undergraduate class. A full view of the compiler's landscape is given later in Figure 3.7.

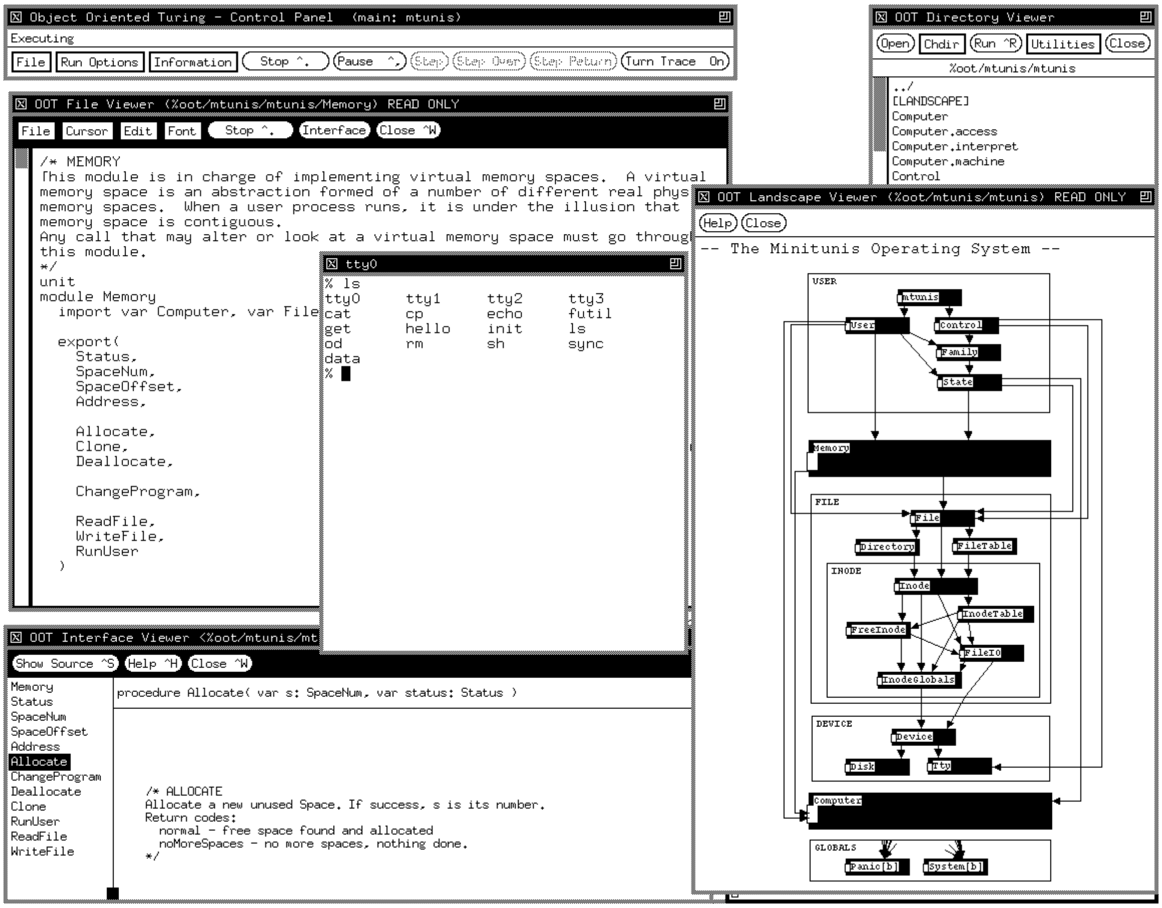

Figure 3.1: **The OOT Program Development Environment:** This figure is a snapshot of the OOT environment. On the top left corner resides the Control Panel. Under this is an OOT File Viewer containing source code. Under the File Viewer is the Interface Viewer which shows the interfaces of a module. The OOT Directory Viewer is at the top right corner and contains a list of the files that make up a system. Below the Directory Viewer is the Software Landscape Viewer which graphically depicts the structure of the system. The central window, labelled tty0, is a run-time window which shows the output generated by running OOT system.

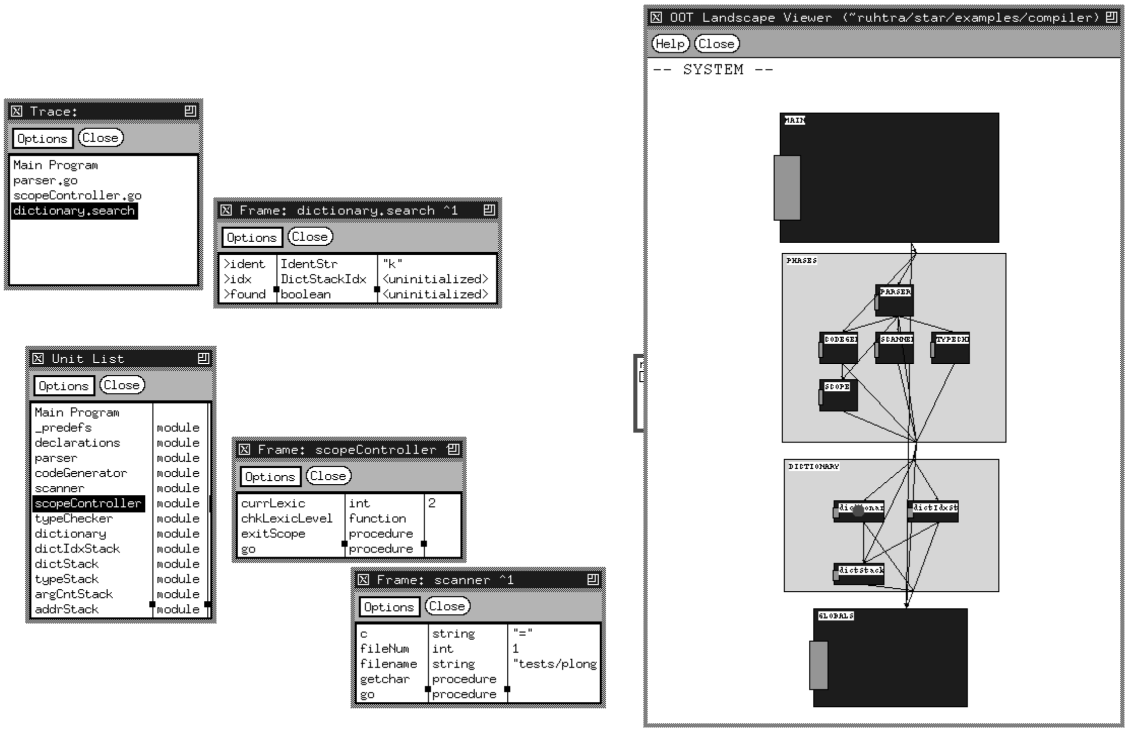

Figure 3.2: **The Run-Time Landscape:** The Landscape Viewer is on the right and shows a hierarchical view of a sample compiler. On the left, the Dataview Tool shows five Dataview windows displaying information an execution stack trace (top-left), a procedure's stack frame (top-right), a list of compilation Units (lower-left), and two units (lower-right and centre).

3.1 Run-Time Artifacts

The run-time artifacts of a software system are based on data values and the call stack. OOT supports lightweight processes (threads) so there are multiple call stacks and data areas. Data values can further be subdivided into statically allocated data and dynamically allocated data (heap storage). The latter are referenced through a chain of pointers or array descriptors rooted somewhere in the global data space or on the call stack. By following these chains, the Run-Time Landscape can view all of the data in a program.

The current Run-Time Landscape has two major components. The first consists of the pictorial, Software Landscape style view showing software structures down to the subprogram level and the second is a textual view capable of displaying data down to the variable level. The pictorial view is called the *Landscape Viewer* and the textual view is called the *Dataview Tool* (See Figure 3.2).

The Landscape Viewer provides an easy to use tool for following the flow of control in a program as well as displaying profiling information. The Dataview Tool is a graphical user interface for browsing the traditional debugger world of stack traces, procedure stack frames, and pointer value dereferencing. The two components serve complementary roles and allow the user to choose the most useful representation at any time.

In the next section, I will describe the Landscape Viewer in detail followed by a section on the details of the Dataview Tool.

3.2 Run-Time Landscape Viewer

The Run-Time Landscape Viewer retains all of the functionality of the viewer from the static Software Landscape while adding profiling information, a moving execution-path indicator, easy to use breakpoints, and integration with the Dataview Tool.

3.2.1 Profiling

Profiling evaluates the distribution of computing time over the program. The 90-10 rule says that for many programs 90% of the time is spent executing 10% of the code. Given Amdahl's Law[1], knowing which 10% of the program represents the lion's share of execution time is very valuable for performance tuning.

The Run-Time Landscape can present three different forms of profiling information. The first is a graphical view of traditional profiling information given by tools like Unix *prof*. The second shows the concentration of CPU use by the program in the recent past by visualizing the CPU usage as a decaying *heat trail*. The third is an immediate feedback mechanism called the *execution marble* which shows the program flow of control as a moving dot on the landscape.

[1] *Amdahl's Law* [Amdahl, 1967] states that the performance improvement to be gained from using a faster mode of execution is limited by the fraction of the time the faster mode can be used. The Speedup ratio can be defined as the ratio [Hennessy and Patterson, 1990]:

$$Speedup = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using enhancement when possible}}$$

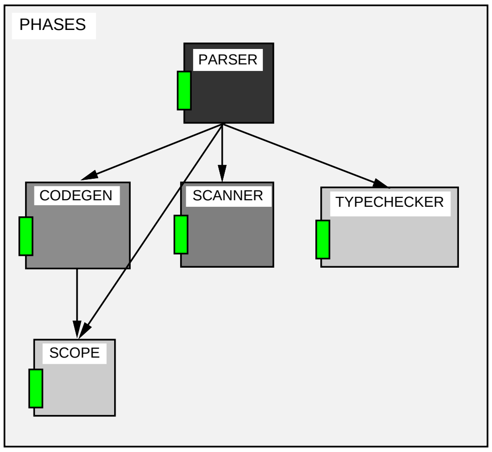

Figure 3.3: **Landscape Profile:** Profile view with different shades used instead of colours to represent the relative execution times. Darker shades represent a higher percentage of CPU usage.

Traditional Profiling

Profiling systems record the execution time of calls and the number of calls to each subprogram. Some profiling systems provide better resolution and generate information for each statement [Eick and Steffen, 1992, Eick et al., 1992]. Profiling information in its raw form consists of execution time measurement data for each call which is often output as a text listing or represented as a simple histogram plot. Profiling information usually comes in two flavours. The first reports a subprogram's execution time for code only in the subprogram. The second form folds in the execution time for all call-graph descendents as well. I will refer to the former as *execution time* and the second as *execution time closure*. Call graph cycles are handled in the same manner as *gprof* where calls into a cycle share the time of the cycle. The Run-Time Landscape Viewer displays qualitative profiling information by showing colour gradations (See Figure 3.3). However, it is sometimes desirable to show quantitative information to solidify the impression given by qualitative information. The Landscape viewer solution for this is to place the time and percentage values next to the box labels in the landscape when requested.

Showing Recent CPU Usage

Another view presents a "heat trail" display. Landscape boxes are coloured in various shades of red to indicate how "hot" they are. The heat trails show how the CPU time has been distributed recently. The analogy would be a person carrying a welding torch and moving among a set of metal boxes to heat them with the torch. The longer he stays with a box, the hotter it gets. A neglected box will slowly cool down. This view needs an adjustment panel available to control the "torch" temperature and the cool down decay rate.

Adding these profiling features to the landscape elision mechanisms allows the user to see the aggregate CPU usage for a whole module or subsystem at a glance. The heat trail display can be useful for visualizing locality of reference in a program's code space.

Moving Execution Marble

The Landscape Viewer shows the flow of control during a program's execution by displaying a circle on the landscape box where program execution currently lies (See Figure 3.4). This circle will henceforth be called an *execution marble* and moves smoothly between landscape boxes with the aid of cartoon animation techniques [Chang and Ungar, 1993]. See also Section 4.7 for implementation details. Multiple threads would be represented by a separate marble per thread in the same manner used by *Paradocs* [Price, 1990].

This *marble animator* on its own can still lead to disorientation if the marble moves about the display too quickly. However, the combination of *execution marbles* with the landscape provides a unique set of features to reduce extraneous movement.

First, the user is able to see the execution of programs as the marble (or marbles for concurrent execution) moves through the landscape. The landscape is no longer a static display. The marble is *live*. It shows the current position of execution in the landscape in real-time. This suggests yet another form of profiling information. The eye tends to follow the marble and note where the marble spends most of its time.

Second, the landscape containment elision allows the user to select varying levels of details. If the details of a subsystem is of no interest, its internal structure can be hidden and the marble will only move to the middle of the subsystem box. This eliminates the fluttering effect that occurs when the marble moves rapidly between subprogram boxes within a subsystem.

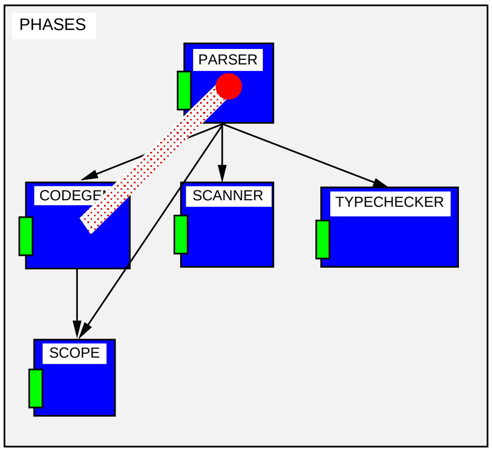

Figure 3.4: **Landscape During Execution:** Marble view shows a marble moving from CODEGEN box to PARSER box to show the flow of control.

Third, the movement of the marble may provide cues for rearrangement of the landscape layout or restructuring of the program. Deep, wasteful calling patterns become evident immediately. Wim De Pauw et al [Pauw et al., 1993] have found some interesting structures by dynamic clustering of objects according to recent object to object communications.

Lastly, the patterns of movement become a rough guide for the user seeking to understand the code. During user testing of *Paradocs* [Price, 1990], Blaine Price found *Paradocs* test subjects narrowed down the problem area better than the control group.

When debugging a system, it is useful to be able to pause at a place in the code. The Run-Time Landscape provides a new method for setting these *breakpoints*.

3.2.2 Landscape Breakpoints

A key ability in debuggers like Unix *dbx* is the setting of breakpoints. A commonly used form of breakpoint asks the debugger to stop execution at the entry point to a given subprogram. At this point the programmer will usually single step through the code while examining data.

The landscape approach to this common operation is the addition of *landscape breakpoints* [Hayes, 1993]. The user presses a key (currently the **z** key) to toggle a landscape box as a breakpoint. This requests that the execution pause at any entry point within the chosen landscape box. When such a breakpoint is encountered during execution, the execution is paused, the source file is shown at the paused line, and the stack trace is shown

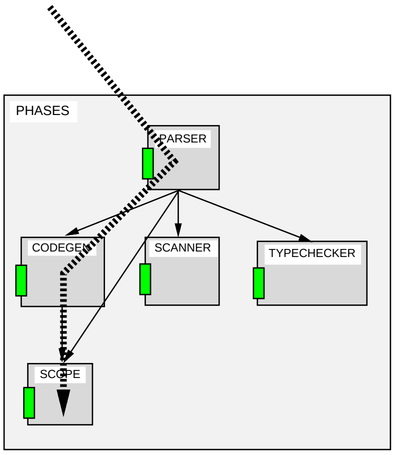

Figure 3.5: **Landscape Stack Trace:** Segmented trace arrow shows the call stack visually.

by arrows in the landscape.

This elegant approach to debugging breakpoints scales well. A user may be testing a new implementation of a module. Setting a breakpoint at the module level is quick, easy, and allows the user to scrutinize each entry point to the new module. A library maintainer would normally group library components into a subsystem. Setting a library breakpoint on the library's subsystem box may be desirable for tracking down a puzzling interaction of library calls with the library's static data.

The Landscape Viewer tool is easy to use for browsing source code and setting breakpoints. The Landscape Viewer also provides convenient access to the Dataview Tool for displaying data associated with Landscape boxes.

3.2.3 Integration with Dataview Tool

There will be times when the abstract views provided by the Landscape Viewer are insufficient for the programmer's needs. For example, a programmer needs to be able to examine the data of the program. The detailed examination of data values is provided by the Dataview Tool. To simplify the transition from the Landscape Viewer to the Dataview Tool, the landscape provides the user with an intuitive method for viewing program data. Simply clicking on a landscape box in data mode requests the Dataview Tool to bring forth a window showing the current data associated with that box.

The next section describes the Dataview Tool in more detail.

3.3 Dataview Tool

The Dataview Tool is a graphical user interface for selecting and displaying the data views of a running program. The information is similar to that provided by traditional debuggers like *dbx* (See Figure 3.2). Incorporated into the Dataview Tool are facilities for keeping displayed data current, simplifying view management, and browsing source code.

View updating, view management, and source browsing will be discussed after the following subsection on Dataview Tool window type.

3.3.1 Introduction to Dataview Window Types

The Dataview Tool controls seven types of windows. Each window type shows a particular class of information and has menu options specific to that type. The set of these windows is called a *view*. Views can be loaded and saved between sessions to speed up the development cycle (view management). The following is a description of each type (See Figure 3.6).

Thread Queue List: The list of queues of concurrent threads. The current threads in a system are divided into groups based on their current state. For example, the currently executing threads are in the *"Running"* queue and threads in the *"Wait on condition* **c***"* queue are those waiting on condition **c**.

Thread Queue: The list of threads in the thread queue. The thread names are presented for further investigation.

Thread Stack Trace: A stack trace for the thread is shown as a list of subprogram names and optionally with the corresponding filenames and line numbers.

Stack Frame: The active variables in the current stack frame with any subprogram parameters are shown. The identifiers' names, types, and values are given.

Symbol: This Dataview window type shows more details on a symbol. For example, it will show a list of fields if viewing a record. A field may be a record type and may be clicked on to show further information. For subprograms, the formal parameters and their types are given. Viewing a pointer dereferences the pointer to show the indirect data that it references whether it be a data type or a class instance. In the case of a class instance, the instance's symbols are revealed along with any associated data much like a Stack Frame display.

Unit List: The names of all separately compiled units.

Unit: This will show the unit's top level symbols including unit variables and subprograms. A Unit type window is essentially the same as a Stack Frame window in functionality. However, a Unit window uses a different path through the Dumper interface (See 4.4) to get the desired information. Normal Stack Frame windows obtain information from the Dumper via the call stack.

3.3.2 View Updating

The currently active set of windows comprises the current *view layout*. During trace mode or after a single stepping operation, all view windows in the current view layout are updated to reflect any changes in the data. Sometimes a view window becomes invalid (e.g. a function terminates thus removing its active stack frame). In such cases, the window is removed from the display but not from the current layout. If and when the view window becomes valid again, the window is shown on the display in the same position and size as before.

3.3.3 View Management

The Dataview Tool provides for the convenient selection of a program's data. View management operations allow the user to save view window layouts and recall them later. This is a convenience for the user to reduce the time taken to find, select, and reposition view windows .

3.3.4 Source Browsing

The original Software Landscape associates files with landscape boxes to allow quick access to files. In the Software Landscape, blue boxes represent code in a file, and pressing **f** on any blue box opens the corresponding file for viewing. The Run-Time Landscape allows quick navigation from the landscape to the Dataview Tool as was mentioned in Section 3.2.3. In a similar manner, the Dataview Tool allows the user to smoothly move between the data views and the associated source code. The user may request to see the file associated with a variable by selecting it and pressing **f**. The file is then opened and positioned where the variable is declared.

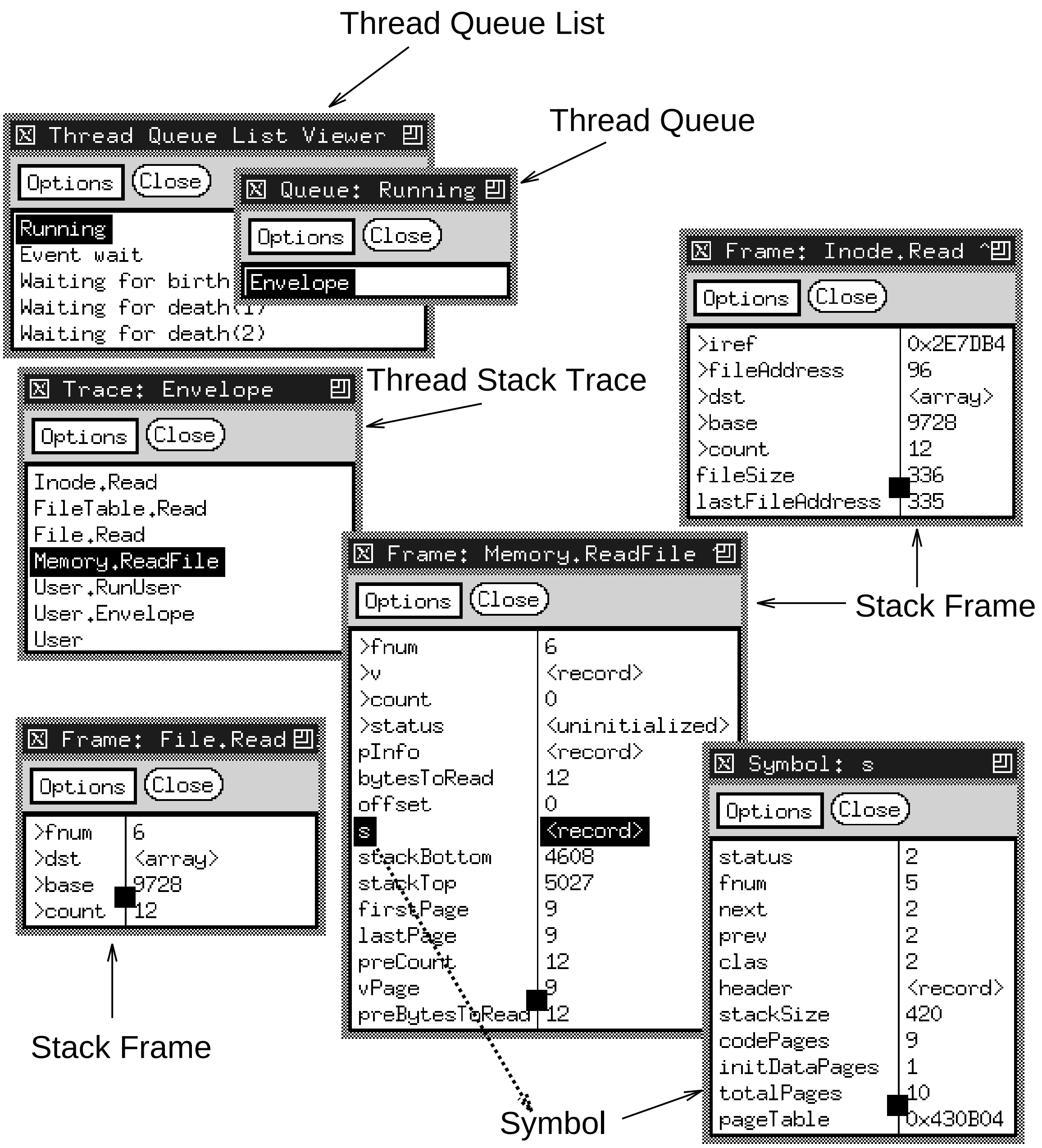

Figure 3.6: **Dataview Tool:** This figure is a snapshot of the Dataview Tool showing the Dataview Tool's window types labelled. The Unit List and Unit window types are not shown in this figure.

3.4 Run-Time Landscapes in Action

In this section, I describe how the user might interact with the *Run-Time Landscape* displays when used for profiling, debugging, and program understanding. All of the following examples are based on a 9000 line compiler written in OOT for a senior undergraduate class. A typical landscape view of the whole compiler is shown in Figure 3.7.

3.4.1 Profiling

The Run-Time Landscape provides profiling in three ways. One is the instantaneous information provided by the execution marble. Observing the marble's movement gives the user immediate feedback on the program's execution activities in the subsystems. The second form of profiling information shows the accumulated execution statistics using a combination of colour shading and quantitative values. The third profiling method is the *heat trail* display showing recent CPU distribution. This section will only consider the use of the first two forms of profiling. For example, Figure 3.8 shows what the landscape looks like when asked for a profile view after running the program on a test file.

This shows the PHASES and DICTIONARY subsystems as the major users of execution time. The output shown in Figure 3.8 is suspect in that the GLOBALS subsystem accounts for a significant percentage of the execution time. The GLOBALS subsystem only contains the **declarations** module where most of the compiler's state variables are declared and initialized.

In OOT, it is common for code to be executed for module initialization. This initialization code usually exists to setup initial data structures or assign values to an array. In this case, the cause for the GLOBALS execution time is a loop to clear the simulated machine's memory. Since this initialization is the same for all program runs, it makes sense to remove it for most profiles. (It may still be useful for tuning the initialization code itself.) Figure 3.9 shows the same diagram compensated for initialization code by use of a menu option. From this view, it is clear that the PHASES subsystem is responsible for almost all of the execution time. The user could zoom into the PHASES subsystem to take a closer look at profiling details within PHASES (See Figure 3.10).

From these displays, it is clear that the parser, scanner, and code generation subsystems are the three major "hot spots." Looking inside those three subsystems reveals relative

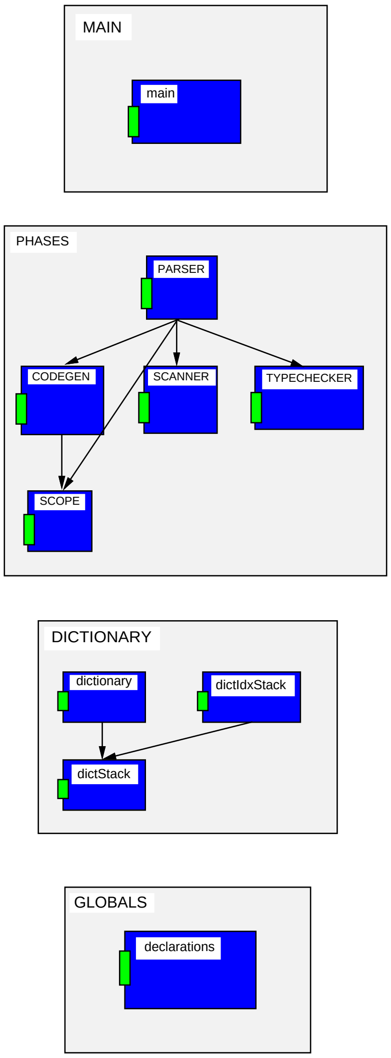

Figure 3.7: **Example Landscape:** The landscape for the compiler example showing the structure of the major subsystem components (MAIN, PHASES, DICTIONARY, GLOBALS). To simplify the diagram, the arrows crossing subsystem boundaries have been left out.

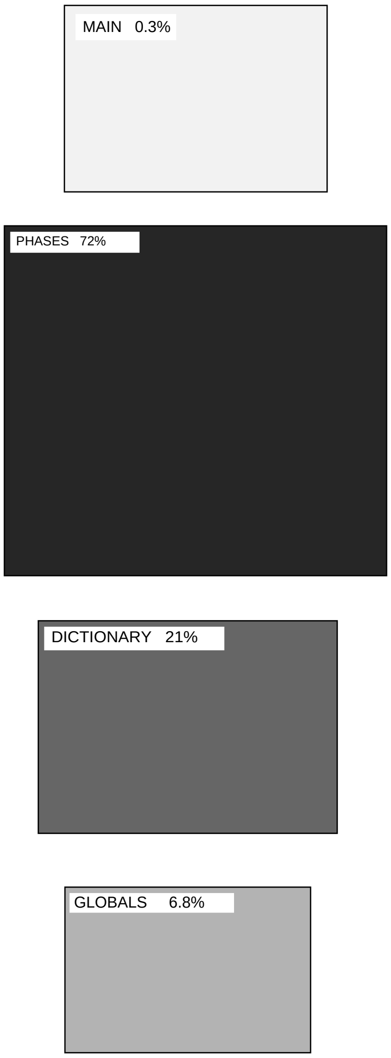

Figure 3.8: **Profiling Example:** This view shows the MAIN, PHASES, DICTIONARY, and GLOBALS subsystems along with their execution profile data represented as shades. Darker boxes indicate more CPU usage. The percentages shown beside the entity names give quantitative profile data.

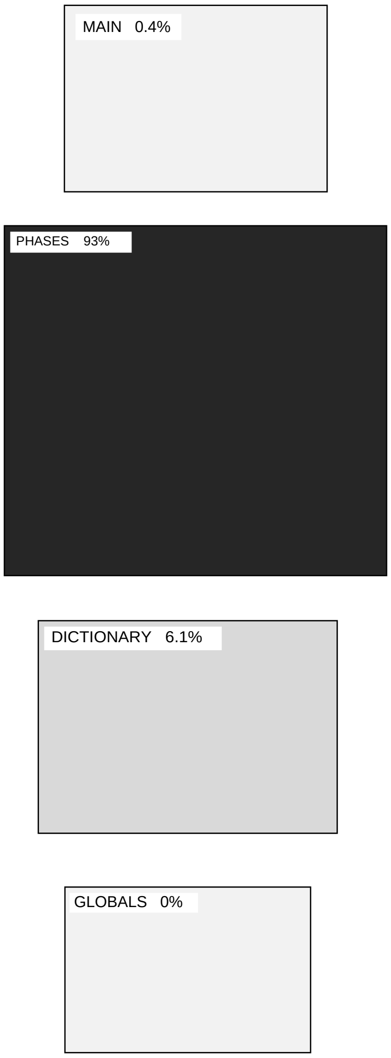

Figure 3.9: **Profile Landscape After Adjustment:** This view shows the MAIN, PHASES, DICTIONARY, and GLOBALS subsystems along with their execution profile data after compensating for initialization execution.

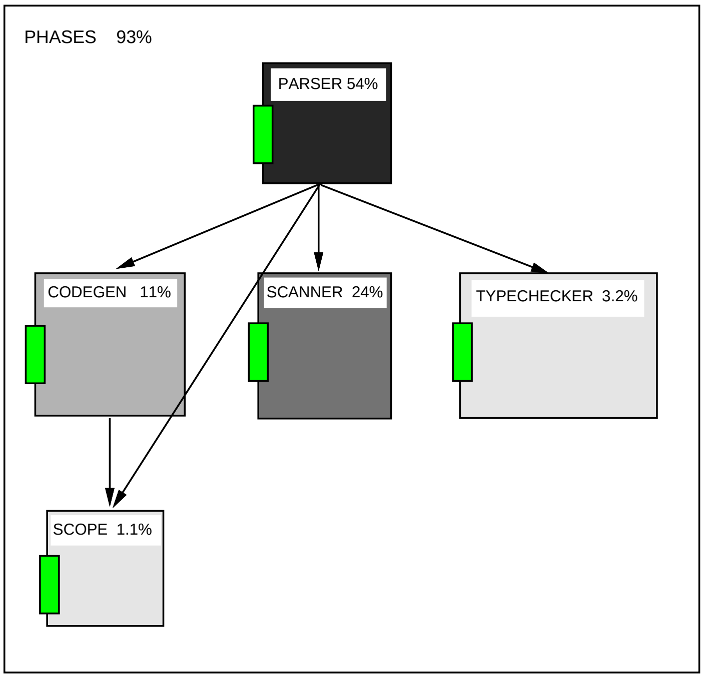

Figure 3.10: **Profile Landscape After Zoom-In:** This view shows the interior of the PHASES subsystem with associated profiling data represented using shaded boxes and percentage values.

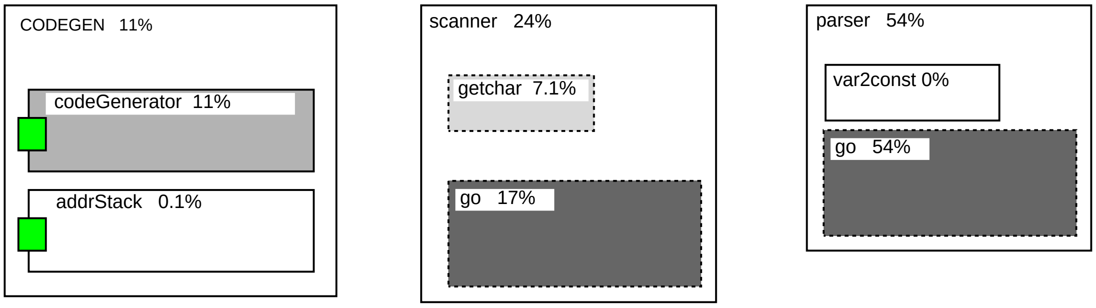

Figure 3.11: **Three More Landscape Zooms:** On the left, the CODEGEN subsystem is opened to show two units where the codeGenerator holds almost all of the CPU usage. The PARSER and SCANNER units in the middle and on the right show internal structure with profiling information. The dashed boxes in PARSER and SCANNER are subprogram boxes. The key feature to note in all the profile views is the consistent model presented to the user from high-level designs down to subprograms.

execution times in finer detail (See Figure 3.11).

This profiling display shows that the parser's *go* procedure is a good candidate for performance tuning. The user can view this procedure by moving the mouse to the box and requesting the file by pressing the **f** key with the pointer over the procedure box. A file window is opened with the parser unit positioned at the beginning of the *go* procedure. Studying *go* identifies a few places for optimization but it is not conclusive.

The scanner module is responsible for tokenizing the input file stream. There are two procedures, *getchar* and *go*. Viewing the source listing for each procedure indicates that *getchar* imposes a steep overhead for each character read and the *go* procedure could be sped up by replacing if-elsif chains with a *case/end case* construct.

A further examination of the codeGenerator module would show the short two line procedure STOMEM as the main culprit in CPU use inside the codeGenerator module. This would be a prime candidate for optimization by inlining the procedure.

This example illustrates how a programmer can isolate CPU "hot spots" and examine them confidently and proficiently. The next subsection shows how one would use the Run-Time Landscape for debugging.

3.4.2 Debugging

There are certain operations commonly performed with traditional debuggers like *dbx* in Unix. Many people begin using *dbx* by showing a stack trace after a memory access violation. Over time, users learn to set breakpoints, single step through the code, and display data values. The efforts to provide debugging features in the Run-Time Landscape have been focused on these most common operations.

Suppose the programmer suspects a problem with the code generator. He pops up the compiler's landscape, moves the mouse to the CODEGEN subsystem, and sets the breakpoint flag for CODEGEN. Soon after beginning to execute, the execution marble will move to the CODEGEN subsystem and stop due to the breakpoint. Popping up a stack trace shows that the codeGenerator's *predefs* procedure is being called. This procedure emits the output code to handle errors and has been checked thoroughly. Hitting the **Resume** button continues execution. The compiler runs a while longer and stops again. The user can click on various displayed objects to examine variables but for this example he decides to single-step through the execution using the **Step** button. In single-step mode, the line of execution is highlighted in an editor window to show the current location of execution (See Figure 3.12). To better understand how the code generator operates, the user pops up data windows showing the address stack and the code output array during execution.

The emphasis of the above scenario is the use of high-level design views integrated with low-level code and data selectors during the compile-execute-debug cycle. The programmer makes active use of the design views in software development thus reducing the likelihood that the design architecture and the implementation diverge.

The next section discusses the use of the Run-Time Landscape in program understanding.

3.4.3 Program Understanding

Software maintenance accounts for a majority of the effort expended on computer software today [Pressman, 1992]. The Run-Time Landscape can help reduce the time spent in *"wheel spinning"* activities like understanding the data structures, performance bounds, interface characteristics, and discovering what the code does.

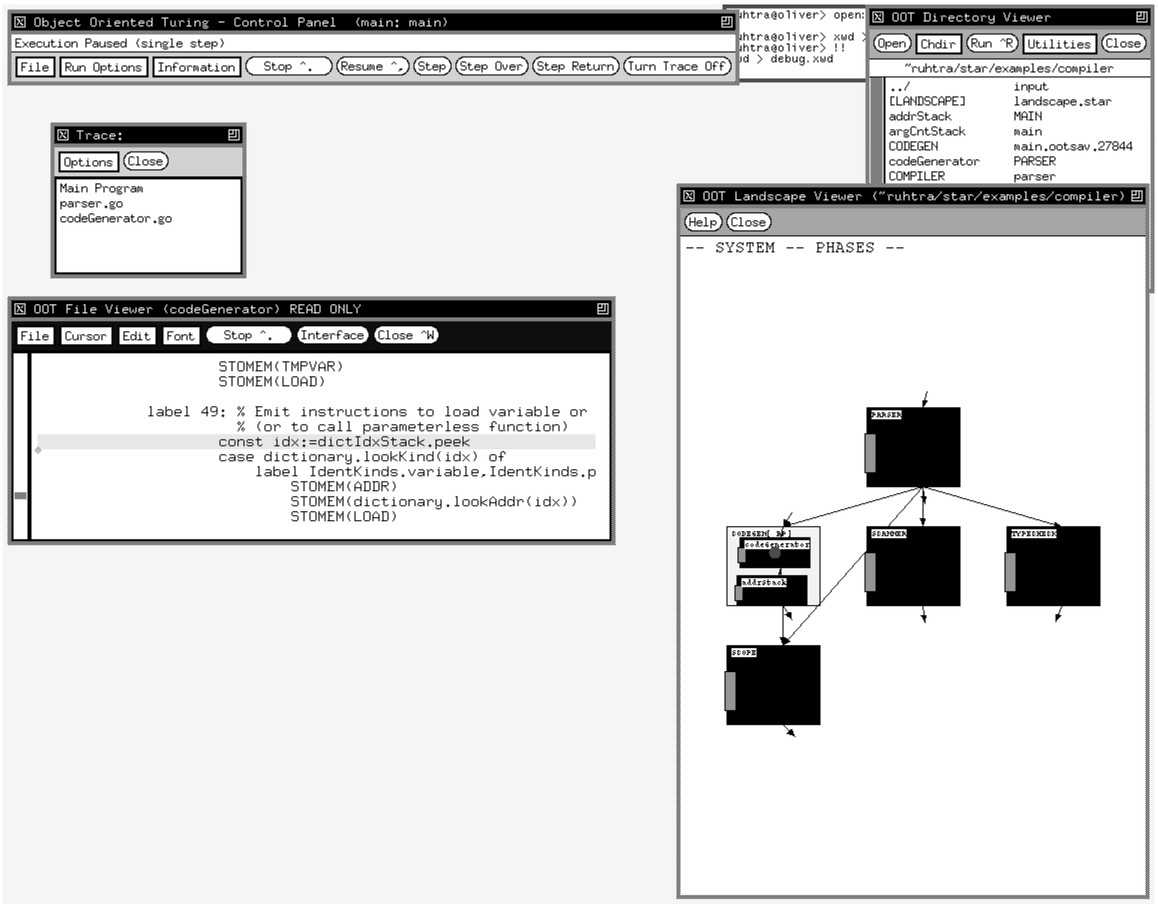

Figure 3.12: **Debugging Session:** This screen snapshot shows an example debugging session using the landscape for breakpoints, a stack trace Dataview Tool window, and an editor highlighting the the current line of execution in single-step mode.

The program understanding benefits of the Run-Time Landscape derive from the integration of Software Landscape high-level software designs, profiling views in the Landscape, data views provided by the Dataview Tool, and execution marble code animation.

High-level code structure can be explored using the Landscape browser. The user can use the Dataview Tool to watch data change as the program runs. The execution marble allows the user to observe calling patterns. Watching the execution marble also provides some qualitative profiling feedback in real time.

Returning to the compiler example, the user quickly identifies the parser as a central point of control of this compiler by using the execution marble. It is also clear that the code generator and type checker use the dictionary frequently. Displaying the dictionary stack and dictionary index stack data reveal what the code generator and type checker are doing when they call the dictionary. In this manner, all of the integrated tools provide information from different points of view to present a picture of the running program to the user.

3.5 Summary

This chapter described the Run-Time Landscape in detail and showed how it could be used. The main features of the Run-Time Landscape are:

- Visual profiling information using colour gradations.
- A moving execution marble to give instantaneous feedback of where code is executing.
- Hierarchical landscape views to reduce complexity when displaying profiling information or showing the execution marble.
- Landscape breakpoints to simplify debugging.
- Data views to show program data.
- Data, source code, and landscape browsers are integrated to ease navigation between tools.

The next chapter discusses the Run-Time Landscape prototype and addresses the technical details involved in its implementation.

Chapter 4

Prototype Implementation

The previous chapter described the Run-Time Landscape in detail. A prototype was implemented to explore and demonstrate the usefulness of the Run-Time Landscape. It has also provided a better basis for generating ideas for future work. This chapter provides details on the subset of Run-Time Landscape features in the prototype and how they were implemented.

4.1 Features implemented

The prototype Run-Time Landscape has a Landscape Viewer with an animated execution marble, landscape breakpoints, and integration with the Dataview Tool. The Dataview Tool supports all of the view window types with displays updated during program execution and a point and click interface to navigate through the run-time data and follow pointer references.

4.1.1 Features not implemented

Unfortunately, the prototype does not implement all of the capabilities envisioned in Chapter 3.

The current prototype does not have elision control in the textual Dataview Tool. There is also no mechanism in place for displaying the contents of arrays although record and union types work.

The marble animator in the Landscape Viewer supports only a single flow of control. In a multithreaded program, the marble animator follows the global flow of control, which exists because concurrency is simulated in OOT, instead of a separate marble following each thread [Price, 1990].

The other major omissions from the Landscape Viewer are profiling displays using colour gradations to show cumulative execution time and the tracking of created and destroyed class instances. The examples shown in the previous chapter represent rough data collected by timer calls added to the prototype.

4.2 Landscape Layout: Using the Star System

The layout of landscapes plays an important role in presenting a comprehensible view of a software system to the user. It was fortunate that a tool to generate and lay out landscapes from source code was already available. The Star System [Mancoridis et al., 1994] is a tool composed of a set of parsers and code generators for converting design relationships between numerous syntactic formats. Each parser extracts relations from a specific notation such as OOT source code into an internal representation of containment and dependency relations that can be output via any of the available code generators. One of the Star System's code generators outputs a landscape file after computing a reasonable layout. Therefore, one of the capabilities of the Star System is the ability to parse OOT programs and produce a landscape layout. These automatically generated landscapes are visually pleasing and correspond exactly to the source code structure by nature of their construction.

The Run-Time Landscape extends Software Landscape views down to the subprogram level. The current version of the Star System's landscape code generator does not supply information lower than modules and classes. The prototype finds the existing subprograms (See Section 4.4) and inserts them into the landscape.

The next section briefly explains the OOT environment internal structure, which served as the software basis for the Run-Time Landscape prototype.

4.3 Interfacing with the OOT Environment

The OOT environment (See Figure 4.1) compiles source code to a pseudo-code (p-code) representation and then executes the p-code with a p-code executor. The p-code instruction set consists of a fairly rich set of instructions for a stack machine architecture. The p-code executor simulates the stack machine architecture required by the p-code. This organization

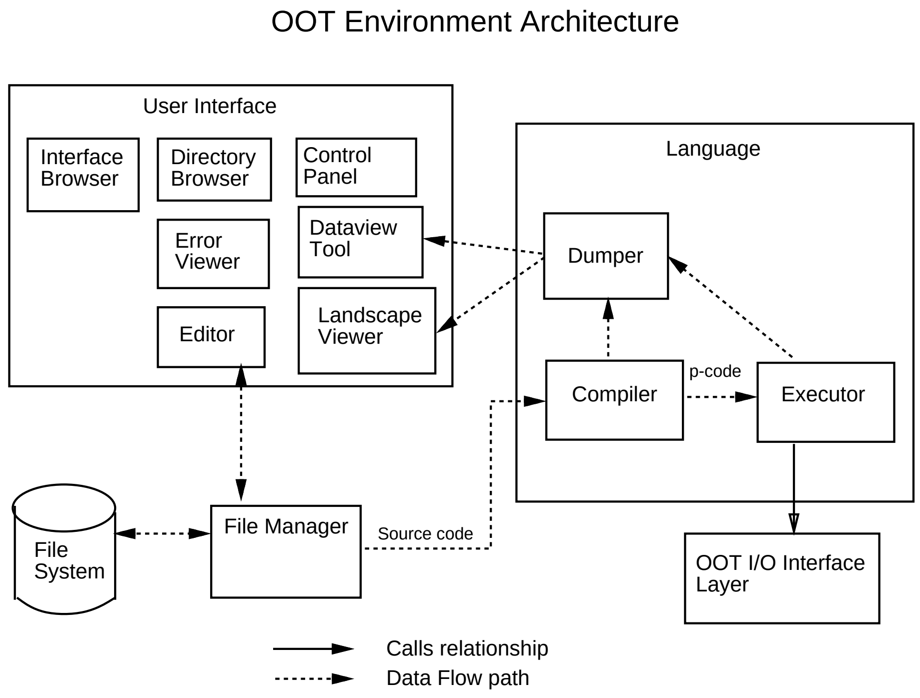

Figure 4.1: **OOT Environment Architecture:** This figure shows the overall architecture of the OOT Environment. The dashed arrows show the data flow from the source code to the compiler and executor through the dumper to the Landscape Viewer and Dataview Tool. (Not all relationships are shown.)

provides a portable p-code compiler/executor combination and fast compilation of OOT programs at the expense of execution speed approximately 100 times slower than natively compiled programs. However, the user gains the benefits of an interactive environment designed for education and for rapid prototyping. During the execution of an OOT program, the environment calls the p-code executor to run some code when not occupied by other housekeeping duties such as updating the display (See Figure 4.2).

When the executor is called, it runs until an error occurs or 2000 p-code instructions have been executed. This is the standard behaviour of the OOT interpreter (See Figure 4.2). The OOT environment checks the executor return values to see if a run-time error occurred. If trace mode is active, other actions may take place such as showing the current line of execution in an editor window. Similar actions are performed if the return from the executor was due to the user requesting single step execution. This is where Run-Time Landscape displays are updated. During a trace or single step mode update, any changed values in Dataview windows are updated and if a Landscape Viewer is present the stack trace for

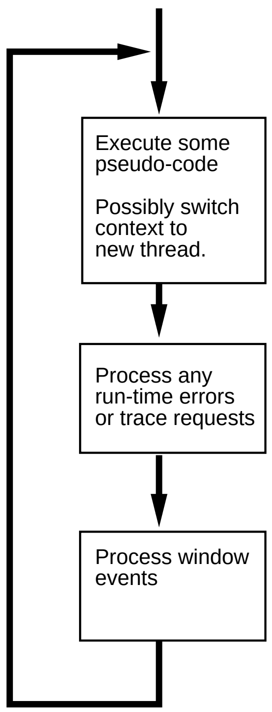

Figure 4.2: **OOT Execution Loop:** This figure shows the program flow during the execution of an OOT program.

the currently executing thread is used to initiate a possible marble motion. The call to initiate a marble motion returns a flag indicating whether or not a breakpoint has been encountered. This method does have one minor flaw. That is, the marble is usually located one line behind the actual execution. In practice, this is not a concern because the marble moves very quickly in trace mode and the movement will keep up with the user while single stepping.

4.3.1 Landscape Breakpoints

Landscape breakpoints are stored as a flag attribute on landscape boxes. I will discuss how this works in Section 4.6. If a breakpoint has been reached during trace mode, execution is paused and a message is displayed to the user to indicate execution was paused by a landscape breakpoint.

The next section explains the method used to extract run-time information and provides details on the *Dumper* interface used.

4.4 Extracting information by probing the environment

The Run-Time Landscape extracts data about a running OOT program by polling the compiler/executor for information (See Figure 4.1). The compiler generates and keeps symbol information and other relevant data accessible in a database called the *Dumper* module.

This has a number of advantages over approaches that report events via code annotations (See Section 2.4). First, polling can be done at an update interval determined by the user. Second, a well-defined, portable interface can be designed to relay information to one or more visualizer clients. This also allows the compiler maintainer to hide the details of the database structure. Third, the private database structure can accommodate tradeoffs in design between maintaining a complete database and computing information when requested for it. For example, the current line of program execution can be handled in several ways. One method has a current position variable updated after each p-code instruction. A more refined approach would be to generate an update for such a variable only when it changes. Another approach takes the current p-code simulator's program counter and finds the current line and file by searching a data structure holding p-code address boundaries. The first two methods slow down execution but allow very fast discovery of the currently executing line. The third method sacrifices information retrieval performance for faster p-code execution. Finally, a well-defined polling interface can be extended more easily for remote access between machines as well as providing a clean and natural separation of programming labour for the creation of the visualization system.

Naturally, there are also disadvantages to a polling approach. First, an extra interface layer to get the information will impose a performance penalty over more direct approaches. This is to be expected but the benefits can overcome the cost. The code is not altered and the performance penalty is paid during updates on only the data under observation. Second, it is more difficult to follow changes to the internal state of a program. A tightly coupled run-time interface would be free to signal changes to specific data locations and for other fine-grained observations. It may be possible to address some of these issues by extending the dumper interface to permit the setting of transition traps and other active feedback mechanisms. Lastly, a polling approach is not suited for generating an exhaustive, comprehensive trace log of execution. Rather, a polling approach is appropriate where information is selectively requested.

The next subsection describes the interface used by the Run-Time Landscape implementation to request information from the OOT compiler/executor.

4.4.1 Dumper Module

The Dumper Module provides an interface for requesting information from the compiler/executor about a running program. The Dumper interface operates by providing some information about the currently running threads. Wherever this starting information is incomplete, a *cookie* is provided. A Dumper Module Cookie is a special index value that can be passed back to the Dumper Module to obtain additional information. Dumper Cookies come in a few varieties such as scope and symbol cookies. Scope cookies point to a particular data scope in the program such as the parameters and local variables in a procedure. Symbol cookies are used to discover more information about a composite type like a record or to dereference a pointer variable.

There are four calls in the Dumper interface.

DumpThreadList() is the normal starting point for data queries. (See figure 4.3). A pointer is returned which references information on thread list categories, their corresponding threads, and the stack trace for each thread. The stack frame records in a stack trace contain the subprogram name, source position, and a scope cookie as information fields. To obtain more information about a stack frame, the corresponding scope cookie is passed to DumpScopeInfo().

DumpScopeInfo() accepts a scope cookie and returns the corresponding scope information available. This is usually called to obtain stack frame information such as the parameters to the subprogram call and the local variable data during the call. (See Figure 4.4). New cookies provide avenues to further examine the subprogram called or variables and data in containing scopes.

DumpSymbolInfo() returns information on the symbol referenced by a provided symbol cookie. This call is used to find more information about complex symbol types or to follow a pointer reference. For example, the cookie associated with a record type will return the fields in the record and the values for each field. Following the symbol cookie for a subprogram name will return the information on the parameters (See Figure 4.4).

DumpUnitList() is a secondary starting point for retrieving run-time information about OOT *units*. A *unit* in OOT is a separately compilable source entity. It may be the case that a program is comprised of several OOT module units and some of the state of the program is contained in module variables. This data may not be accessible because it is neither global nor is it in a containing scope of procedures in the stack trace at the current time (See Figure 4.5). *DumpUnitList* returns a pointer to a list of units represented as symbol records. The scope cookies in these records can be passed to *DumpScopeInfo* to obtain a list of the unit's symbols and their values.

The Dumper Module provides a well-defined interface for obtaining run-time information. The next section discusses how the Dataview Tool provides the user with an easy to use interface for navigating and displaying that information

4.5 Dataview Tool

The Dataview Tool uses readily available components for the X Window System to provide a window-based, interactive data browser. Simple list widgets are used to display the textual data presented by the Dataview Tool. Various options are provided for navigation via

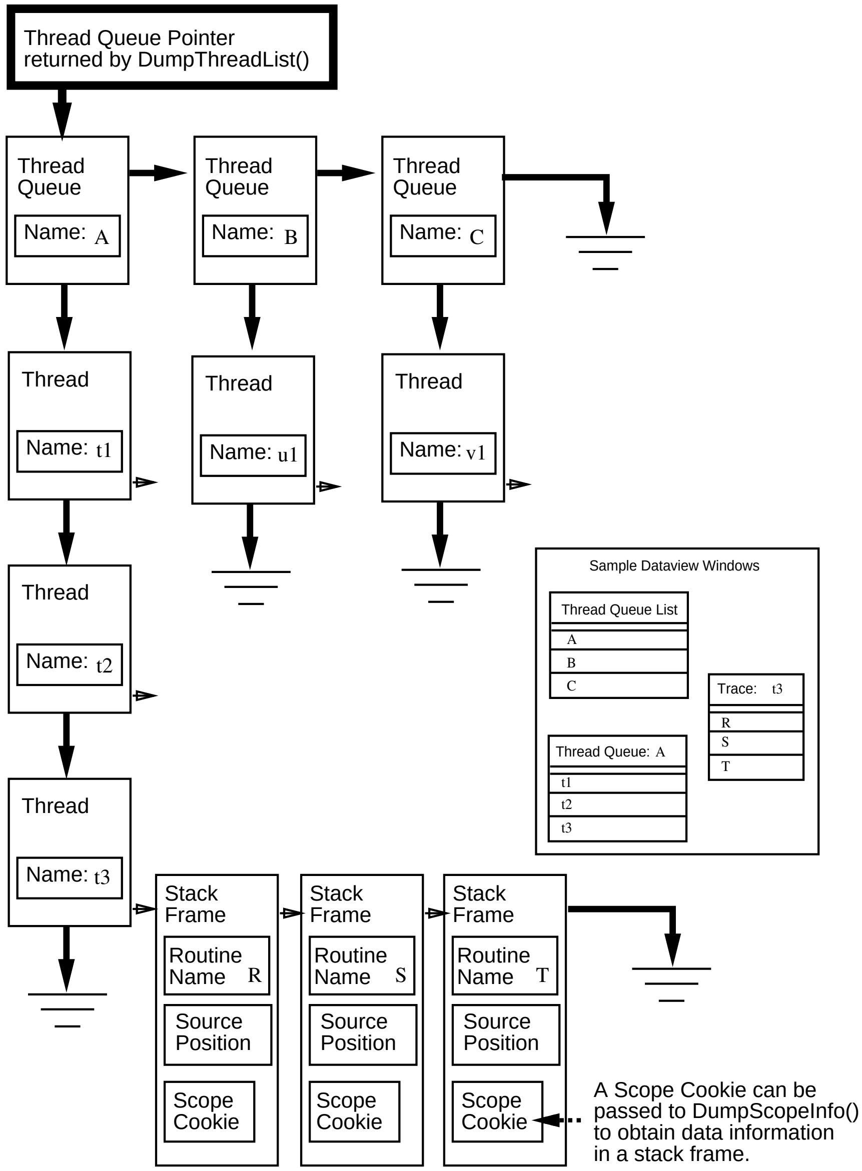

Figure 4.3: **DumpThreadList Data Structure:** This figure shows the data structure returned by a call to DumpThreadList(). The arrows represent pointers to records and the ground symbol is a nil pointer. The small box to the right labeled *Sample Dataview Windows* shows how the data structure names relate to the Dataview Tool.

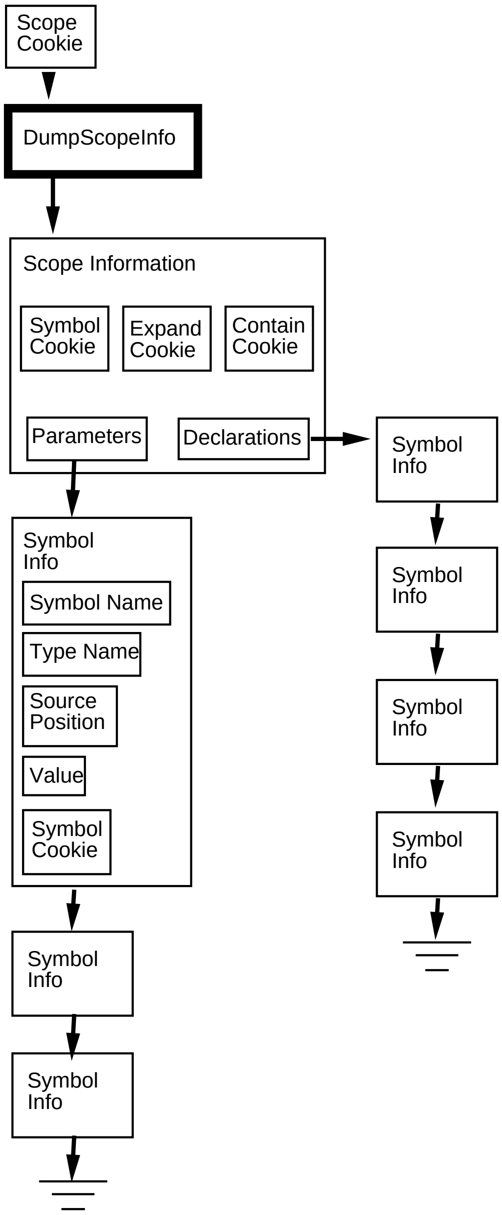

Figure 4.4: **Scope and Symbol Information Data Structures:** This figure shows the data structure returned by DumpScopeInfo(). The structure of Symbol Information records is also shown. At the top of the diagram, the DumpScopeInfo label with a thick box around it represents the function call DumpScopeInfo(). The diagram shows a Scope Cookie input to the DumpScopeInfo() call and the data structure below being returned. The other arrows represent pointers to records and the ground symbol is a nil pointer.

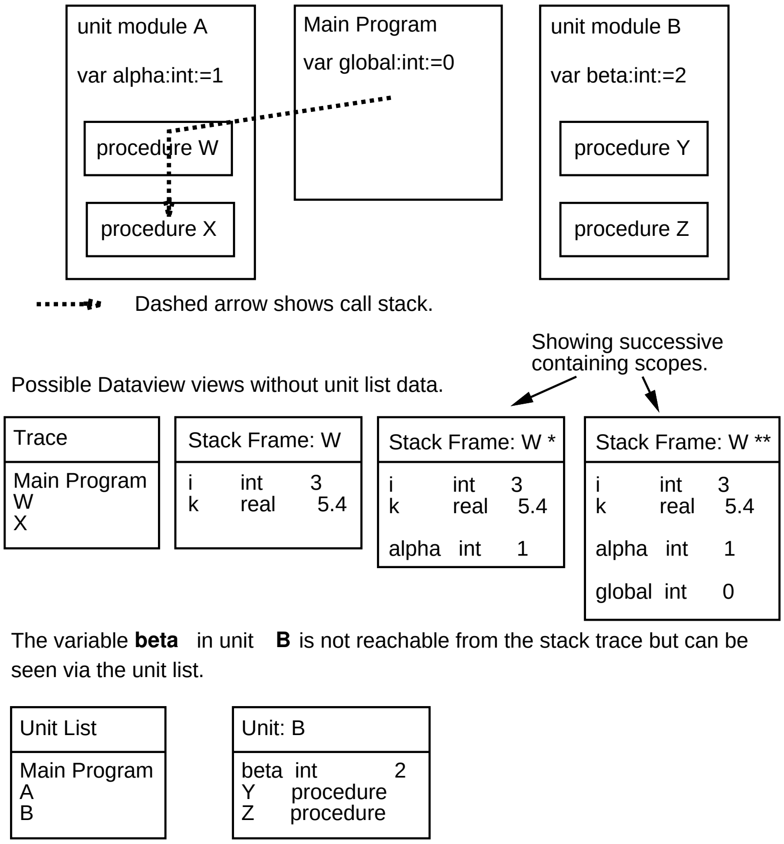

Figure 4.5: **Unit Data:** This figure illustrates a simple program with two units, **A** and **B**. Each unit contains a module variable and two procedures. The main program has called procedure **A.W** which then called **A.X**. The first set of Dataview windows show that the global variables and the module variables in **A** are reachable. However, **B.beta** cannot be reached without the unit list provided by DumpUnitList() in the Dumper Module.

menus, keypresses, and mouse clicks. Despite being a fairly straightforward implementation, there are a few noteworthy points to mention. Namely, the display update mechanism, distinguishing calls of a recursive procedure, and minimizing screen distractions.

4.5.1 Updating a Dataview Window

A Dataview update routine is called after each line of execution in Trace Mode. All of the view's windows are kept in a linked list and the Dataview update routine simply runs an update operation on each window in the list. Therefore, it suffices to explain how an update works on a single view window.

A view window has various attributes and state information of which the *data path* is the most important. A data path is a description of how to navigate the data space tree using Dumper module calls to locate the desired set of information. These descriptions are stored using strings and associated counts to make saving and restoring views possible since pointer references are not guaranteed to remain the same between runs. It is necessary to use textual descriptions since Dumper module cookies are not guaranteed valid across multiple program runs because they contain pointers to internal data in the executor. If a data path cannot be fully traversed, then the corresponding view is skipped and considered to be invalid. The prototype will delete such windows from the display. Another possibility is to darken/grey the window to show it is not active. When a view window is deleted from the display the view window's configuration still persists with its associated *data path* so it can be displayed again when it becomes active. The current window position and size is saved so the user does not need to reposition the window.

As an example, consider a simple stack trace display of a thread named *producer*. Given that a Dumper module search always begins with the thread queue list, we need to know the thread queue name. In this case, it is the *Running* queue and the thread name is *producer* to distinguish it from a thread named *consumer*. It should be remembered that all of the *data path* information is generated by mouse selections made by the user during use of the Dataview Tool. In the above example, the user would double click on the *Running* queue in the thread queue list window to get the *Running* thread queue window and then double click on the *producer* thread in the thread queue window.

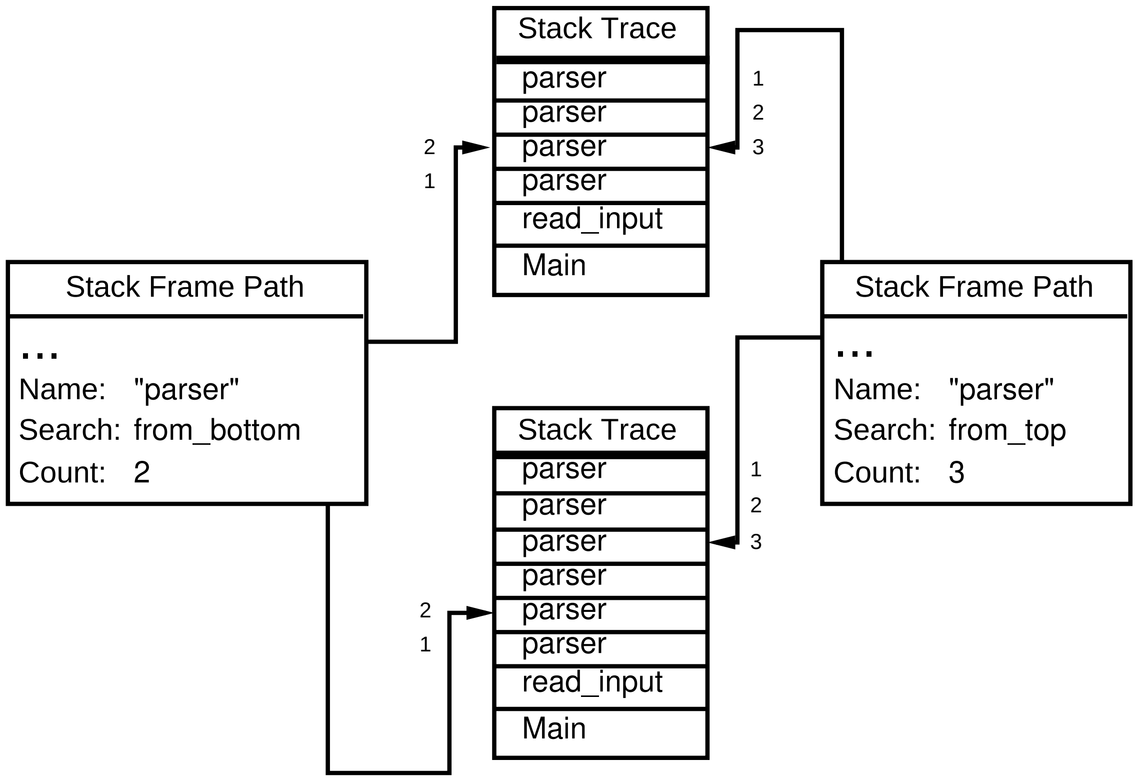

Figure 4.6: **Stack Frame Path Descriptors:** On the left and right are two different stack frame paths. There are two example stack trace instances down the middle. The arrows show which stack frame would be selected by the corresponding stack frame path when given the two example stack traces.

4.5.2 Recursive Procedures

There is one difficulty with this approach. What happens when you try to distinguish among several stack frames generated by a recursive procedure *parser*? Does the user wish to see the deepest calls or the top-most calls? This is where the possible counts in the *data path* descriptor come in (See Figure 4.6). If the user picks the second *parser* for expansion, then the count value will be **2** to remember that this Dataview window was interested in the second *parser* on the list. The counts are normally started from the bottom of the stack to match the common case of looking at the stack frames of a recursive call in the order of the generated calls. The code to count from the top of stack is implemented. A menu option can be used to choose between the two preferences.

4.5.3 Minimizing Screen Distractions

To provide an effective user interface, it is important to reduce flicker and jumpy updates so that only the changing parts of the data display are noticed. This *Principle of Minimal Change* [Holt, 1993] requires designing a user interface that changes as little as possible visually because the human eye is particularly adept at detecting changes and is distracted by them. The Athena widget set provides a list widget that can only be updated *en masse*. No item by item random replacement is provided. Hence, the whole window will tend to blink when the list contents are cleared and updated. An early version of the prototype updated the contents of each list widget on every update cycle even if nothing had changed. This caused all of the Dataview Tool windows to flicker rapidly during execution. This blinking was very distracting because it was unclear where changes occurred. A compromise was to do a brute force, string by string comparision of the currently displayed list and the new list from a view update. If there was no change, the list widget contents were not updated and no blink occurred. A better solution would have a replacement list widget that permitted individual list items to be updated without affecting the whole window.

The next section discusses how the original Software Landscape prototype was adapted for use in the Run-Time Landscape.

4.6 The Landscape as a Repository

The Software Landscape [Penny, 1992, Mancoridis et al., 1993a] prototype written by David Penny represents the landscape diagrams in an elaborate tree structure representing the containment relations with various lists for arrows and back pointers. His prototype provided a convenient repository for storing landscape breakpoint flags and profiling data. The marble animator needs to traverse this tree to identify the box corresponding to a particular procedure name.

To make accesses by the marble animator more efficient, a hash table is generated to map the subprogram names to tree node pointers in the landscape. Given a pointer to a box entity (one type of tree node), a logarithmic traversal up the tree towards the root will show if any containment elision hides the initial box. Such a containing box is referred to as the container and may be the initial box itself if no entity hides it. The container box's width, height, and position can be accessed and the marble uses the middle of the box as a target coordinate for animation purposes. For the animation itself, the marble's current, start, and end positions are maintained along with any in-transit state.

To add breakpoints, a boolean attribute was added to box entities. When the user sets a landscape breakpoint, the attribute is set to *true*. A breakpoint is cleared by setting the flag to *false*. Currently, the 'z' key is used to toggle landscape breakpoints. The same upward traversal used for finding a box's container will show if there is a hierarchical breakpoint flag set on a containing entity.

The next section discusses the techniques employed to produce the smooth marble animation effects.

4.7 Animation of Marble Movement

This section describes in detail how the Run-Time Landscape implements an apparently smooth moving marble using cartoon animator techniques.

4.7.1 Motivation

The marble animator was implemented as a method for showing program flow. The original scheme showed the location of program control by displaying the appropriate landscape box in a different colour. The result was a visualization where boxes rapidly changed colour. It looked interesting but not enlightening because the lack of directional cues overwhelmed the user with flashing boxes that were hard to follow. Animation techniques such as motion blur seemed ideally suited to control flow visualization [Chang and Ungar, 1993]. The hope was that even a simple version of blurred animation would reap most of the same benefits as complete implementations by providing enough information for the human observer to synthesize the illusion of motion.

4.7.2 Animation Phases

Cartoon animators present the following phases to help viewers follow the action in an animation [Thomas and Johnston, 1981, Chang and Ungar, 1993].

Anticipation is a short movement *opposite* to the direction of travel. This brief cue signals the viewer that motion is going to take place.

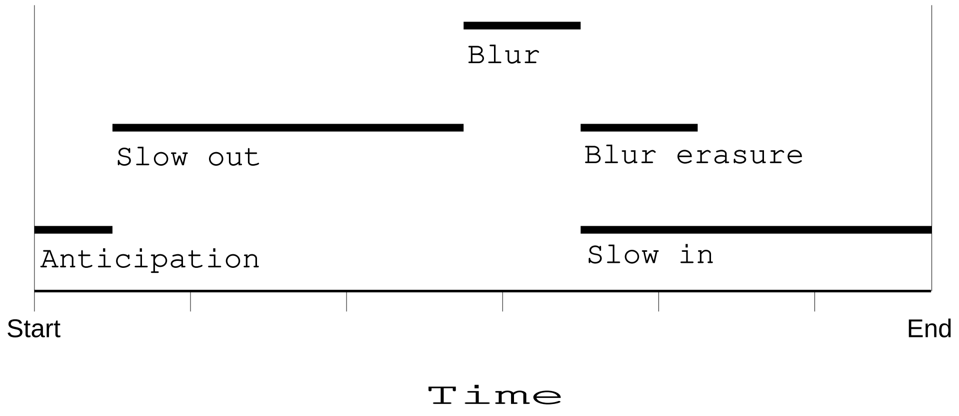

Figure 4.7: **Animation Timeline:** The diagram shows the relative start times and durations of the marble animation phases.

Slow out is the acceleration phase of motion. The direction of motion becomes clear during this phase.

Blurring is used in the middle of a fast motion animation to emphasize both the direction and speed of the object.

Slow In is the deceleration phase. This phase is often paired with **Slow out** and referred to as the *Slow In/Out* phase because they are essentially equivalent except for direction of time.

Arcs are used instead of straight line movement to produce a livelier and more engaging motion.

Follow through is the addition of extra cues after the moving object has just stopped. For example, a target will wiggle after being hit by an arrow.

The marble motion animation in the *Run-Time Landscape* employs anticipation, slow in/out, and blurring. The current prototype does not produce any arc or follow through effects. Figure 4.7 shows a timeline for a full movement sequence of the execution marble.

4.7.3 Real-time Animation

It is desirable to maintain a low overhead in processing and complexity for the animation code. To implement blurred animation for interactive use I used colourmap animation to non-destructively overlay the marble graphics over the landscape pictures. This technique removes the need to intertwine the marble animation code with the landscape drawing code as well as requiring no costly and distracting redraw operations for the landscape.

32 colours (5 bits) are allocated for the landscape in such a way that one common bit location in each pixel value partitions the 32 colours into two disjoint sets of 16 pixel values. One set of 16 pixels is assigned the colours used by the landscape drawings. The other set is assigned red for every pixel value and is used for marble animation. This effectively produces a 4-bit layer for drawing the landscape diagrams and a 1-bit layer for drawing the marble animation. The X Window System supplies *plane masks* which can be used to limit all drawing to a specific set of bits in each pixel (See Figure 4.8). By setting a plane mask to the above mentioned common bit, drawing and erasing operations will appear as though on a glass sheet over the landscape display. There is a significant performance advantage to this approach since no redrawing from a backing store or other means occurs when a marble or its trail is erased. Figure 4.9 illustrates the technique just described.

4.7.4 Slow movements

Small movements of the marble are accomplished by drawing the marble at the destination location. The marble is designed as a circle of "on" pixels surrounded by a thin ring of "off" or background pixels. This means that redrawing the marble only a few pixels offset from the previous location will completely obliterate the old marble. The plane mask prevents any effect on the landscape drawing. This simplifies the small movements in the anticipation, slow in, and slow out phases because the marble doesn't need to be erased at each step. Figure 4.10 shows how this works.

4.7.5 Coordinating the Phases

The above describes how the drawing of the marble can be performed independent of the landscape drawing. This greatly simplifies the structure of the animator. Now, I will describe the steps taken as each phase is drawn. Each step is normally triggered by a timeout event at regular intervals unless they are invoked in rapid succession in Catch-Up Mode which I will describe shortly in Section 4.7.7. Forward and reverse are defined in the natural way. Forward means along the line from the start position towards the end position. Reverse is in the opposite direction along the same line. Movement along the line in single

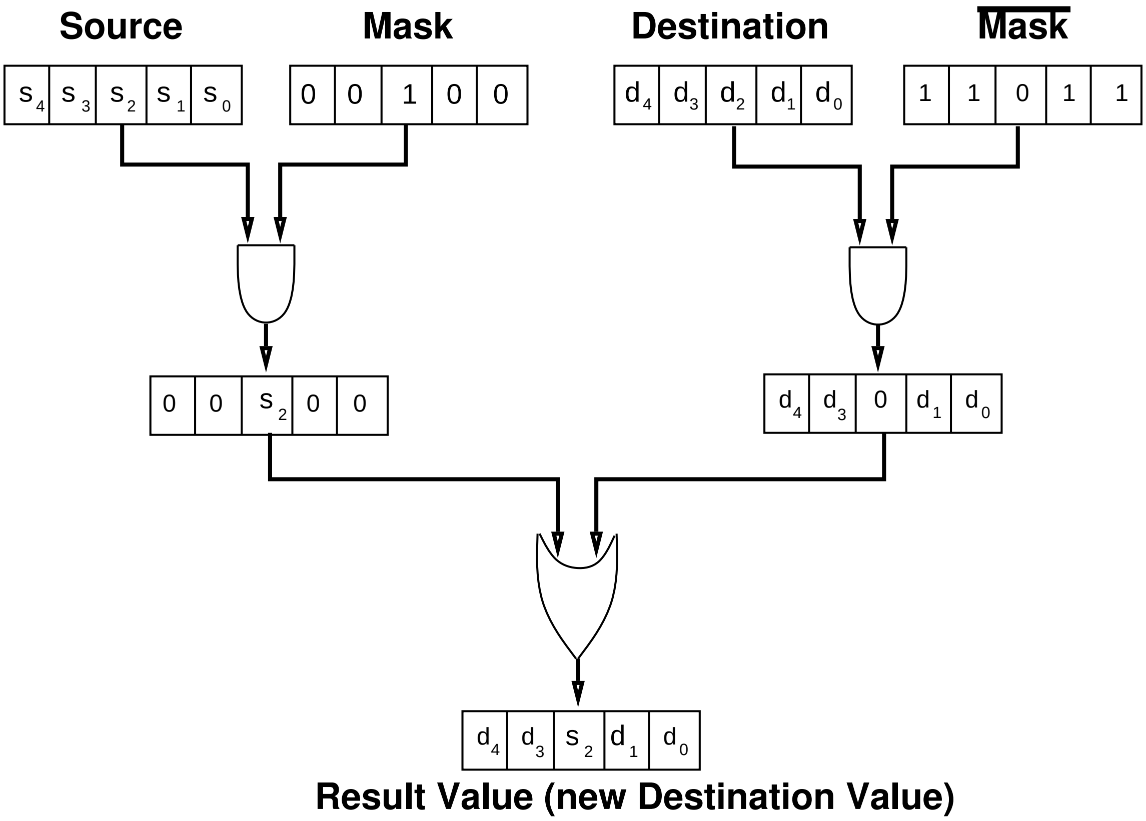

Figure 4.8: **Plane Mask Logic:** This figure shows the assignment of a source pixel value to a destination pixel value when a plane mask is used.

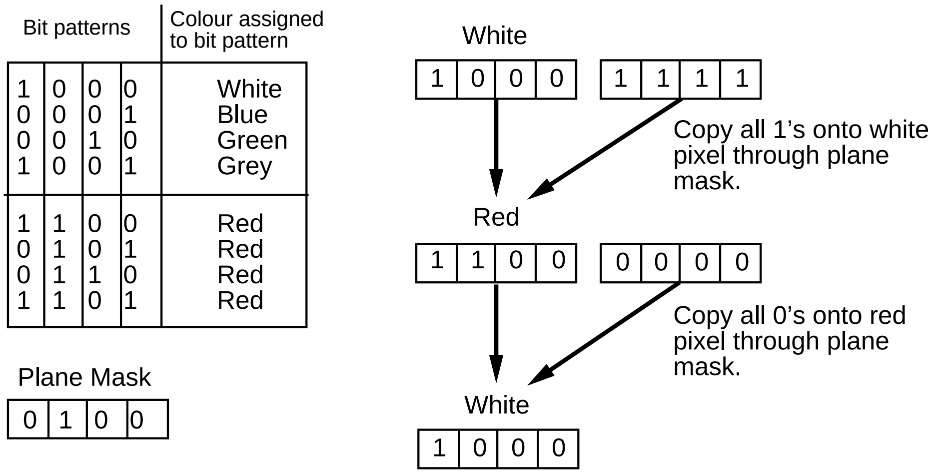

Figure 4.9: **Overlays Using Bitplane Masks:** An illustration of how bitplane drawing masks can be used to create overlays. This example shows a 4 colour plane and a 1-bit overlay plane existing in a 4-bit graphics environment.

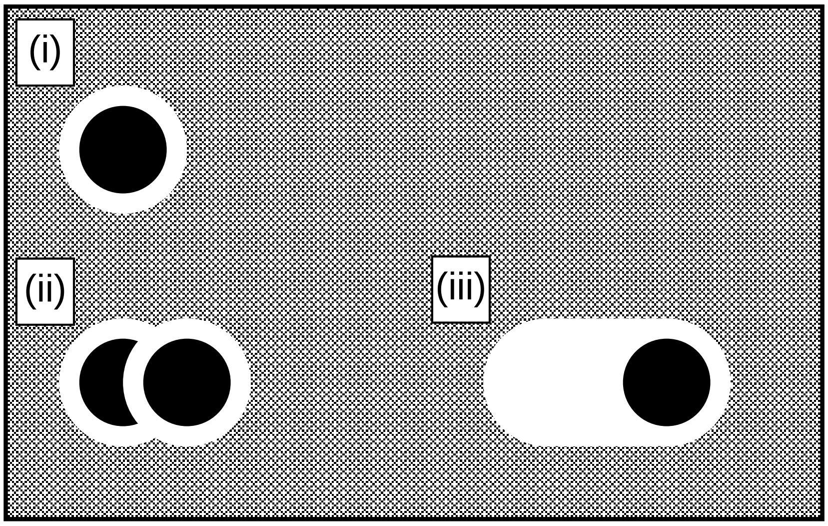

Figure 4.10: **How Small Marble Movements Are Drawn:** The shaded background helps to illustrate the erasing pixel boundary. **i)** The marble is drawn once. **ii)** The result of drawing two overlapping marbles. The marble on the right was drawn after the left marble but offset to the right. It was offset too far to completely erase the first marble. **iii)** The result of drawing a sequence of nine marbles with each drawn only slightly to the right of its predecessor. When seen, the marble would appear to be moving smoothly to right.

steps correspond to the discrete coordinates computed by the Bresenham line rendering algorithm. All coordinates for all phases are computed before the marble animator begins the first phase to show anticipation.

The *Anticipation* phase draws the marble n_{antic} steps in reverse and then n_{antic} steps forward to the start position again.

The *Slow out* phase continues to show forward movement by drawing the marble n_{acc} steps forward.

The *Blurring* phase erases the marble and draws n_{blur} wide, stippled lines to show a streak from (almost) the start to the end position. Each step in the blur phase draws the next blur segment. At the same time as the last blur segment is drawn, the marble is drawn near the end position ready to be moved into its final position in the next phase. The blur segments are erased in the slow-in phase (See Figures 4.12, 4.13, and 4.14).

The *Slow In* phase has two concurrent responsibilities. Primarily, it draws the marble in $n_{deceleration}$ steps to reach the final position. Simultaneously, the blur segments are erased in the same order as they were drawn (See Figures 4.15, 4.16, and 4.17).

The first version of the marble animator had no anticipation, no slow in or out phases, and the blur phase was composed of only one long blur segment. It was surprising how effective such a simple implementation could be. This motivated the more elaborate, multiple phase scheme implemented shortly thereafter in the prototype. The current design provides a smoother animation but there is definitely a diminished return for the additional effort. Some of the phases are shortened or skipped depending on the travel distance in pixels on the screen. For instance, the blur phase is normally composed of three blur segments but there may be fewer blur segments for shorter distance movements.

4.7.6 Drawing blurs

The blur image is implemented by drawing wide, stippled line segments. The stippling combined with the bitplane mask allows much of the landscape diagram to *"show through"*. The width is set to approximate the width of the marble. A normal blur sequence is composed of three contiguous blur segments (See Figures 4.12, 4.13, and 4.14). The middle segment is much longer than the start and end segments to further the illusion of acceleration and deceleration.

4.7.7 Marble-Program Execution Consistency

As mentioned earlier, this implementation uses a timeout event to space out the animation sequence over a period of time (usually about 0.5 seconds). After the initial setup and computation of movement coordinates is made, a timer event is initiated to call a timeout handler. The timeout handler uses the information to walk along a simple state machine to perform the required steps in order.

Trace mode still allows fast enough execution that the marble would fall behind in the animation (0.5 seconds is a long time during the execution of a program). It would be preferable to have a smooth degradation in detail as in Artkit [Hudson and Stasko, 1993] since temporal consistency is an important human factor. The compromise reached was the addition of a Catch-Up Mode. If a marble movement is requested before the completion of the previous movement, Catch-Up Mode is begun. Currently, all this means is that all of the steps in a movement sequence are performed in rapid succession without the delays of a timeout event. It would be possible to omit steps or shorten phases in a more intelligent manner but this prototype doesn't implement them. The net effect is reasonable since the

Figure 4.11: **Marble Blurring:** Start position.

blurring effect dominates when the marble moves very quickly. The full smoothness of the animation engine is really only visible when manually single-stepping through a program.

4.8 Summary

This chapter details the interesting aspects of the Run-Time Landscape prototype implementation. The key areas of the prototype discussed are:

- The use of an abstract interface, the Dumper module, for retrieving run-time information via non-invasive probes.
- The Dataview Tool's management of data views and locating the data in those views.
- Animation effects used for the execution marble and the graphics techniques to implement them efficiently.

The next chapter ends the thesis with concluding remarks and avenues for further research.

Figure 4.12: **Marble Blurring:** First step in blur. Blur segments are drawn using a wide, stippled line.

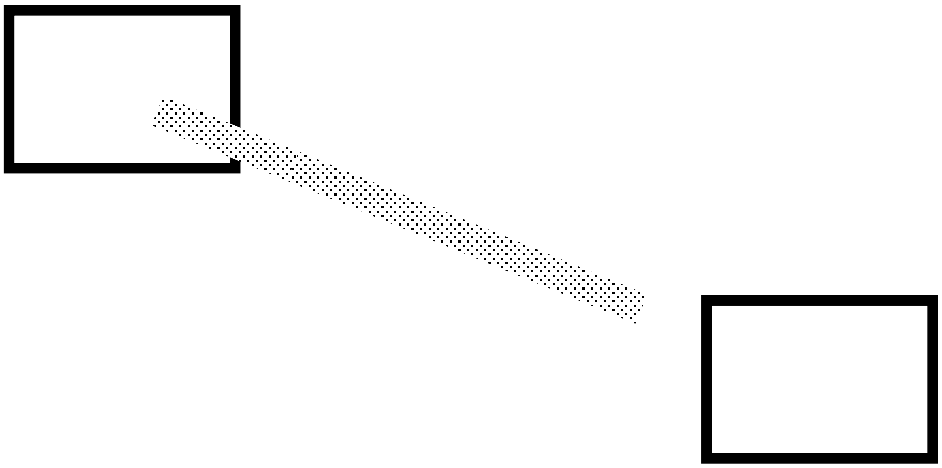

Figure 4.13: **Marble Blurring:** Second step in blur.

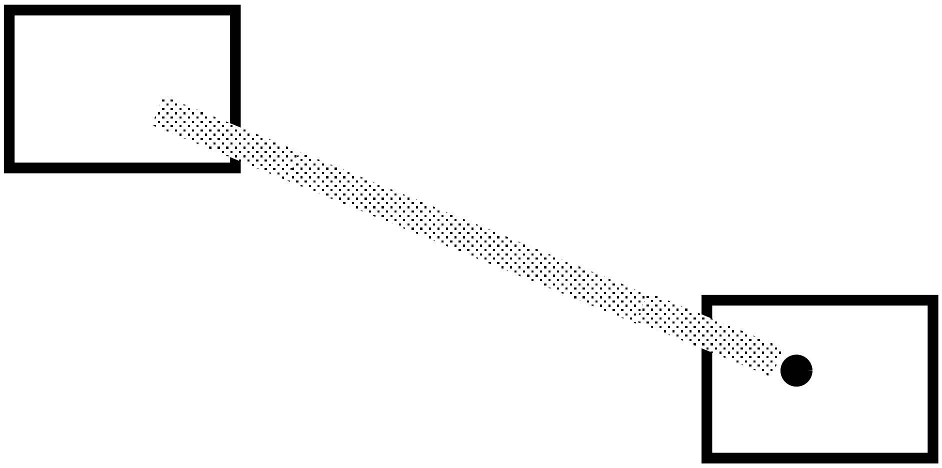

Figure 4.14: **Marble Blurring:** Third step in blur. Marble is placed at beginning of slow in phase.

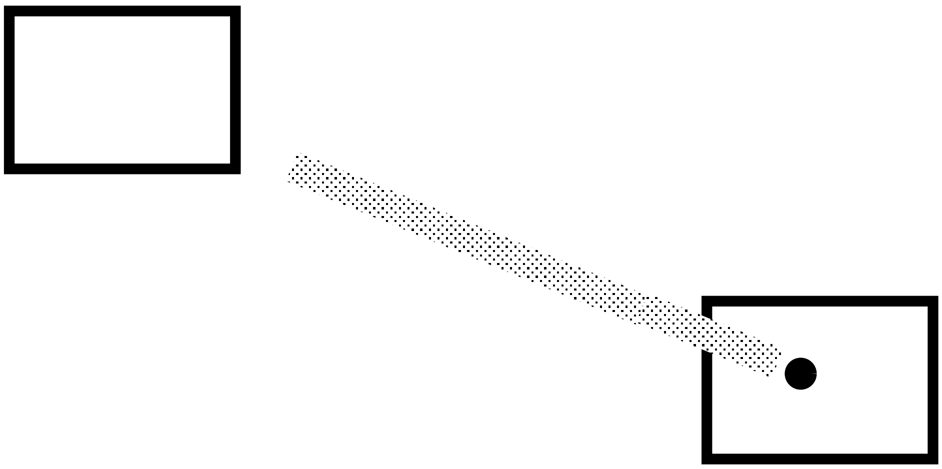

Figure 4.15: **Marble Blurring:** Slow-in movement of marble simultaneously with fade of blur.

Figure 4.16: **Marble Blurring:** More blur faded.

Figure 4.17: **Marble Blurring:** Blur has faded. Marble is almost in final position.

Figure 4.18: **Marble Blurring:** Final position.

Chapter 5

Conclusions

This final chapter summarizes the main points of the Run-Time Landscape and discusses possible directions for further research.

5.1 Recapitulation

The first chapter of this thesis described the motivation for visualization systems. The Run-Time Landscape was introduced as a system for viewing a software system's static and dynamic information for use in profiling and debugging. Hierarchical containment filtering and cartoon animation effects are an important part of the Run-Time Landscape since they provide the means to manage a program's volume of information.

The background information presented in Chapter 2 provided a context to place the Run-Time Landscape among prior systems. The Run-Time Landscape is an automatic program visualization system which displays a program's information while the program is executing. Manual systems have a great deal of difficulty scaling to large systems while other automatic systems tend to present too much information without adequate filtering tools. The Run-Time Landscape uses animation effects to increase the cognitive bandwidth to the user while providing Software Landscape hierarchical views as an interactive filtering mechanism.

Chapter 3 gives a detailed description of the Run-Time Landscape and how it would be used for tasks such as debugging and profiling. Landscape views, profiles, marble animations, breakpoints, and Dataview windows are presented as a tightly integrated package that provides a programmer with helpful tools for software browsing, debugging, and understanding the run-time behaviour of the program.

The technical issues involved in the creation of the prototype are explained in Chapter 4. What is most noteworthy about the prototype implementation are the methods used to produce animation effects efficiently and the use of a well-defined interface for obtaining program information. The Dumper Module information interface provides a clean abstraction for the movement of Run-Time Landscapes onto other platforms using different languages and compilers.

5.2 Contributions

The basic idea of the Run-Time Landscape was to explore software designs using a visual model of execution. The resulting evolution of the Software Landscape shortens the semantic gap between programming-in-the-large and programming-in-the-small by showing that landscapes scale down to the subprogram level and provide a single tool for both high-level software design and low-level debugging and profiling. Closing the gap between design and implementation will help ensure that the design documents and the source code remain mutually consistent during software development.

The work done by Baeza-Yates on VCC [Baeza-Yates et al., 1992] demonstrates that automatic program visualizations are possible in compiled environments like C. A weakness of VCC is its lack of effective filters for displaying and navigating a program's run-time information. The Run-Time Landscape uses several methods to increase the information bandwidth to the user (colour and animation) and filter extraneous data (hierarchical views). Systems without automated extraction such as BALSA [Brown and Sedgewick, 1984] cannot scale up to large software systems with the ease of automated systems like the Run-Time Landscape.

One strength of the Run-Time Landscape is its generality. Two early considerations were that information be extracted automatically and the core system design should work with other languages under both interpretive and compiled environments. The abstract interface of the Dumper Module gives the Run-Time Landscape the possibility of migrating beyond OOT and interpretive environments. The VCC system annotates and collects information from C programs sufficient for the needs of the Run-Time Landscape. All that is required is a suitable Dumper Module interface implementation to enable the use of Run-Time

Landscapes for C programs.

In brief, the Run-Time Landscape presents an automatically generated visual model of execution and utilizes effective presentation methods to show program data and control behaviour without making sweeping assumptions about the language or execution environment.

5.3 Future Directions

The Run-Time Landscape has been an exciting development with many potential uses. However, there are still several areas which could be explored further.

5.3.1 Dataview Tool

The Dataview Tool user interface still needs much work. For example, a nice feature would be the use of colour gradations to highlight changed data values according to how recently they were changed. This would be a form of *heat trail* display for data values. Another area of interest is the use of pictures in the display of data like those in Amethyst [Myers et al., 1988]. The user interface for browsing arrays is another problem that needs to be addressed.

5.3.2 Generalization

The ideas presented in the Run-Time Landscape are not confined to OOT or interpretive execution. As mentioned above, VCC demonstrates that one can automatically annotate a C program to provide the same run-time information desired by the Run-Time Landscape in a compiler based environment such as C. It would be interesting to see Run-Time Landscapes used in other development environments to broaden exposure and to discover shortcomings in the Landscape design.

5.3.3 Class Instances

One major problem still underexplored is the problem of class instances. Namely, how should class instances be presented in the landscape? One line of thought expresses the belief that each class instance should become a first class object in the landscape replicated from the parent class' diagram. This approach lends itself to instance rearrangment and the clustering views by Wim De Pauw et al [Pauw et al., 1993]. However, the resulting landscape may become too cluttered to use effectively. Another possibility adds small "growths" to the class box in the landscape for each instance thus maintaining a tight coupling of instance to parent class. Whatever the outcome, it is becoming increasingly clear that object oriented programs need new and better ways to cope with the dynamic structure of their class instances because the static view can fall far short of clearly expressing the dynamic structure.

Bibliography

[Amdahl, 1967] Amdahl, G. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS 1967 Spring Joint Computer Conference*, Atlantic City, N.J.

[Baecker, 1986] Baecker, R. M. (1986). Panel Discussion: An Application Overview of Program Visualization. In *ACM Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'86)*, page 325.

[Baecker and Sherman, 1981] Baecker, R. M. and Sherman, D. (1981). Sorting out Sorting. 25 minute 16mm narrated colour film. Presented at ACM SIGGRAPH '81.

[Baeza-Yates et al., 1992] Baeza-Yates, R. A., Jara, L., and Quezada, G. (1992). VSS: Automatic Animation of C Programs. In *COMPUGRAPHICS'92, Lisbon, Portugal.*

[Borning, 1981] Borning, A. (1981). The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387.

[Borning, 1986] Borning, A. (1986). Defining Constraints Graphically. In *ACM Conference on Human Factors in Computing Systems (CHI'86)*, pages 137–143.

[Brown, 1988] Brown, M. H. (1988). Exploring Algorithms Using Balsa-II. *IEEE Software*, 21(5):14–36.

[Brown and Sedgewick, 1984] Brown, M. H. and Sedgewick, R. (1984). A System for Algorithm Animation. In *ACM Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'84)*, volume 18,3, pages 177–186.

[Chandhok et al., 1991] Chandhok, R., Garlan, D., Meter, G., Miller, P., and Pane, J. (1991). *Pascal Genie, 1.0, Pascal programming environment with integrated structure-driven editor running on Macintosh computers.* Chariot Software Group, San Diego, CA.

[Chang and Ungar, 1993] Chang, B.-W. and Ungar, D. (1993). Animation: From Cartoons to the User Interface. In *ACM SIGGRAPH User Interface Systems and Technology (UIST'93)*, pages 45–55.

[Consens, 1994] Consens, M. P. (1994). *Creating and Filtering Structured Data Visualizations Using Hygraph Patterns.* PhD thesis, Department of Computer Science, University of Toronto.

[Consens et al., 1993] Consens, M. P., Hasan, M. Z., and Mendelzon, A. O. (1993). Debugging Distributed Programs by Visualizing and Querying Event Traces. In *Extended Abstract in Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging. See also Univ. of Toronto Tech Report CSRI-285.*

[Consens and Mendelzon, 1993] Consens, M. P. and Mendelzon, A. O. (1993). Hy^+: A Hygraph-based Query and Visualization System. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Washington, DC.

[Eick and Steffen, 1992] Eick, S. G. and Steffen, J. L. (1992). Visualizing Code Profiling Line Oriented Statistics. Technical Report Unknown: eick@research.att.com, AT&T Bell Laboratories.

[Eick et al., 1992] Eick, S. G., Steffen, J. L., and Jr., E. E. S. (1992). Seesoft – A Tool For Visualizing Line Oriented Software Statistics. Technical Report Room IHC 1G-339, AT&T Bell Laboratories.

[Furnas, 1986] Furnas, G. W. (1986). Generalized fisheye views. In *ACM Conference on Human Factors in Computing Systems (CHI'86)*, pages 16–23.

[Glinert and Tanimoto, 1984] Glinert, E. P. and Tanimoto, S. L. (1984). Pict: An Interactive Graphical Programming Environment. *IEEE Computer*, 17(11):7–25.

[Goldstein and von Neumann, 1947] Goldstein, H. and von Neumann, J. (1947). Planning and coding problems of an electronic computing instrument. In Taub, A., editor, *von Neumann, J., Collect Works*, pages 80–151. McMillan, New York.

[Haibt, 1959] Haibt, L. M. (1959). A Program to Draw Multi-Level Flow Charts. In *The Western Joint Computer Conference*, pages 131–137, San Francisco, CA.

[Hayes, 1993] Hayes, W. (1993). Private Discussion: On Landscape Breakpoints.

[Hennessy and Patterson, 1990] Hennessy, J. L. and Patterson, D. A. (1990). *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, California.

[Henry et al., 1990] Henry, R. R., Whaley, K. M., and Forstall, B. (1990). The University of Washington Program Illustrator (UWPI). In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 223–233.

[Holt, 1993] Holt, R. C. (1993). Private Discussion: On Principle of Minimal Change for GUIs.

[Holt and Cordy, 1988] Holt, R. C. and Cordy, J. R. (1988). The Turing Programming Language. *Communications of the ACM*, 31(12):1410–1423.

[Holt et al., 1992] Holt, R. C., Penny, D. A., and Mancoridis, S. (1992). Multicolour Programming and Metamorphic Programming: Object Oriented Programming-in-the-Large. In *Proceedings of the 1992 IBM CASCON Conference*, pages 43–58.

[Hudson and Stasko, 1993] Hudson, S. E. and Stasko, J. T. (1993). Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions. In *ACM SIGGRAPH User Interface Systems and Technology (UIST'93)*, pages 57–67.

[Letovsky, 1988] Letovsky, S. (1988). Plan analysis of programs. Technical Report Research Report No. 662, Yale University.

[London and Duisberg, 1985] London, R. L. and Duisberg, R. A. (1985). Animating Programs Using Smalltalk. *IEEE Computer*, 18(8):66–71.

[Lutz, 1992] Lutz, R. (1992). Plan diagrams as a basis for understanding and debugging pascal programs. In Eisenstadt, M., Keane, M., and Rajan, T., editors, *Novice pro-*

gramming environments: explorations in human-compiter interaction, pages 243–285. Lawrence Erlbaum Associates, East Sussex, UK.

[Mancoridis, 1992] Mancoridis, S. (1992). A Visual Programming Environment for Object-Oriented Turing. Master's thesis, Department of Computer Science, University of Toronto.

[Mancoridis et al., 1994] Mancoridis, S., Holt, R. C., and Godfrey, M. W. (1994). A Program Understanding Environment Based on the "Star" approach to Tool Integration". In *Proceedings of the Twenty-Second ACM Computer Science Conference*.

[Mancoridis et al., 1993a] Mancoridis, S., Holt, R. C., and Penny, D. A. (1993a). A Conceptual Framework for Software Development. In *Proceedings of the Twenty-First ACM Computer Science Conference*, pages 74–80.

[Mancoridis et al., 1993b] Mancoridis, S., Holt, R. C., and Penny, D. A. (1993b). A "Curriculum-Cycle" Environment for Teaching Programming. In *Proceedings of the Twenty-Fourth ACM SIGCSE Technical Symposium on Computer Science Education*, pages 15–19.

[Mendelzon, 1993] Mendelzon, A. O. (1993). Declarative Database Visualization: Recent Papers from the Hy+/Graphlog Project. Technical Report CSRI-285, University of Toronto.

[Mukherjea and Stasko, 1993] Mukherjea, S. and Stasko, J. T. (1993). Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding. In *15th IEEE International Conference on Software Engineering (ICSE'93)*.

[Myers, 1986] Myers, B. A. (1986). Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *ACM Conference on Human Factors in Computing Systems (CHI'86)*, pages 59–66.

[Myers, 1988] Myers, B. A. (1988). The State of the Art in Visual Programming and Program Visualization. Technical Report Technical Report CMU-CS-88-114, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

[Myers, 1990] Myers, B. A. (1990). Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1(1):97–123.

[Myers et al., 1983] Myers, B. A., Chandhok, R., and Sareen, A. (1983). Incense: A System for Displaying Data Structures. In *ACM Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'83)*, pages 115–125.

[Myers et al., 1988] Myers, B. A., Chandhok, R., and Sareen, A. (1988). Automatic Data Visualization for Novice Pascal Programmers. In *IEEE Workshop on Visual Languages*, pages 192–198.

[Pauw et al., 1993] Pauw, W. D., Helm, R., Kimelman, D., and Vlissides, J. (1993). Visualizing the Behaviour of Object-Oriented Systems. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 326–337.

[Penny, 1992] Penny, D. A. (1992). *The Software Landscape: A Visual Formalism for Programming-in-the-Large.* PhD thesis, Department of Computer Science, University of Toronto.

[Pressman, 1992] Pressman, R. S. (1992). *Software Engineering: A Practitioner's Approach, 3rd Ed.* McGraw-Hill, New York, NY.

[Price, 1990] Price, B. A. (1990). A Framework for the Automatic Animation of Concurrent Programs. Master's thesis, Department of Computer Science, University of Toronto.

[Price, 1992] Price, B. A. (1992). Exploring Intelligent Automatic Software Visualization. Ph.D. Qualifying Examination Report, Human Cognition Research Lab, The Open University, Milton Keynes, UK.

[Price, 1993] Price, B. A. (1993). Research Proposal: Intelligent Automatic Software Visualization. Human Cognition Research Lab, The Open University, Milton Keynes, UK.

[Price et al., 1993] Price, B. A., Baecker, R. M., and Small, I. S. (1993). A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4:211–266.

[Price et al., 1992] Price, B. A., Small, I. S., and Baecker, R. M. (1992). A Taxonomy of Software Visualization. In *25th Hawaii International Conference on System Sciences*, volume II, pages 597–606. IEEE Press.

[Sarkar and Brown, 1992] Sarkar, M. and Brown, M. H. (1992). Graphical Fisheye Views of Graphs. In *ACM Conference on Human Factors in Computing Systems (CHI'91)*, pages 83–91.

[Sarkar et al., 1993] Sarkar, M., Snibbe, S. S., Tversky, O. J., and Reiss, S. P. (1993). Stretching the Rubber Sheet: A Metaphor for Viewing Large Layouts on Small Screens. In *ACM SIGGRAPH User Interface Systems and Technology (UIST'93)*, pages 81–91.

[Shilling and Stasko, 1992] Shilling, J. J. and Stasko, J. T. (1992). Using Animation to Design, Document and Trace Object Oriented Systems. Technical Report GIT-GVU-92-12, Georgia Institute of Technology.

[Shu, 1988] Shu, N. C. (1988). *Visual Programming.* van Nostrand Reinhold, New York, New York.

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

[Thomas and Johnston, 1981] Thomas, F. and Johnston, O. (1981). *Disney Animation: The illusion of Life.* Abbeville Press, Ney York.

[Wills, 1990] Wills, L. M. (1990). Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence*, 45:113–171.

[Wills, 1992] Wills, L. M. (1992). Automated Program Recognition by Graph Parsing. Technical Report AI-TR 1358, MIT Artificial Intelligence Laboratory.