Lazy SETL Debugging with Persistent Data Structures

by

Zhiqing Liu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Computer Science Department
New York University
November, 1994

Approved:
Professor Jacob T. Schwartz

© Zhiqing Liu
All Rights Reserved 1994

To my parents

吴业云 **and** 刘国俊

Acknowledgments

This is a good opportunity for me to thank the people who have contributed to the thesis. It is difficult to imagine that I could complete my research work without the contributions of these people.

I am greatly indebted to my research advisor, Jack Schwartz. He has been my mentor and motivator since I started to work with him three years ago. His interest, insight, inspiration, enthusiasm and encouragement have long been part of the driving force of my research work. He has made detailed comments of this thesis, from wording to its overall structure.

I would like to thank David Bacon. He was very kind in explaining to me the details of his SETL quadruples upon which my system is built, and answered many of my questions about and beyond the language. Most importantly, due to his passion to SETL, I have come to share his belief that SETL is a most wonderful programming language.

I would like to thank many people who have commented or criticized my research and thesis. Ed Schonberg, who has encouraged and supported me during my stay at NYU, carefully examined my work and provided invaluable input. Bob Paige, another reader, suggested many improvements in a short period. Ken Perlin discussed with me many interface issues and served on my committees in various occasions. I would also like to thank Robert Dewar, Malcolm Harrison, Alan Siegel, Ben Goldberg, Jiawei Hong, and Xiaonan Tan for their comments about my work.

Finally, I would like to thank the support from my family over the difficult period of the last five years. This thesis is dedicated to them.

Table of Contents

Acknowledgments v

Table of Contents vi

List of Figures vii

List of Tables x

CHAPTER 1 Introduction 1

1.1 The Problem 1
1.2 Summary of Research 3
1.3 Overview of Dissertation 5

CHAPTER 2 Debugging Issues 6

2.1 General Debugging Issues 6
2.2 Related Research 8
2.3 Lazy Debugging 19

CHAPTER 3 A LSD Debugging Example 22

3.1 Using the Debugger 22
3.2 The Main Window 23
3.3 Loading a Debugging Target 24
3.4 Program Execution 25
3.5 The Input and Output Streams 26
3.6 The History Window 26
3.7 The Stack Window 28
3.8 Printing Variable Values 30
3.9 Program Animation 32
3.10 Simulation of Conventional Debugging Facilities 35
3.11 Debugging Example, Continued 36
3.12 Condensed Execution Histories 37

CHAPTER 4 Persistent Runtime System 40
4.1 General Ideas of Persistent Runtime Systems 40
4.2 System Design 44
4.3 Implementation 50

CHAPTER 5 More on User Interface Design 57
5.1 Overall Design Principles 57
5.2 Execution Information Structures 58
5.3 Execution Trace Display 59
5.4 Interactive Display 60

CHAPTER 6 Internal Structure of the System 63
6.1 Design Considerations 63
6.2 Communication Protocol 64
6.3 Implementation Issues 67

CHAPTER 7 Performance 71
7.1 Performance Analysis 71
7.2 Usability Evaluation 79
7.3 Summary 81

CHAPTER 8 Open Issues 82
8.1 Current Limitations and Possible Solutions 82
8.2 Summary and Comments on Open Research Issues 87

Appendix A References 90

Appendix B Test Programs 97

List of Figures

FIGURE 2-1: A detailed debugging model 7
FIGURE 2-2: A debugging model for use of print statements 10
FIGURE 2-3: A debugging model for breakpoint debugging 11
FIGURE 2-4: Lazy debugging separates debugging from program execution 19
FIGURE 3-1: The main window 23
FIGURE 3-2: Load file 24
FIGURE 3-3: Run program 25
FIGURE 3-4: The input and output streams, showing a normal program termination 27
FIGURE 3-5: The history window, showing an execution history 28
FIGURE 3-6: Zoom effect in the history window 29
FIGURE 3-7: The stack windows, showing program variables 30
FIGURE 3-8: SETL values are treated as trees in printing 31
FIGURE 3-9: Interactive displaying 32
FIGURE 3-10: Examples of variable printing 33
FIGURE 3-11: Animation snapshot 34
FIGURE 3-12: Animation of the SETL expression data*3 35
FIGURE 3-13: Locating a bug 37
FIGURE 3-14: A portion of a coarse execution history 38
FIGURE 3-15: Refining a coarse history 39
FIGURE 4-1: Structures of the SETL persistent runtime system 48
FIGURE 4-2: Breaking a large data object into several pieces 49
FIGURE 4-3: Data representation of stack nodes and their major methods 51
FIGURE 4-4: Data representation of frame nodes and their major methods 52
FIGURE 4-5: Data representation of heap nodes and their major methods 53
FIGURE 4-6: Data representation of variable nodes and their major methods 54
FIGURE 4-7: Structures of the memory management component 55
FIGURE 5-1: Organization of execution history 58
FIGURE 5-2: Efficient execution trace display algorithm 59
FIGURE 5-3: Data representations of variable values 61
FIGURE 6-1: Debugging routines layer as a stub in persistent runtime system 64
FIGURE 6-2: Command protocol 65
FIGURE 6-3: Response protocol 66
FIGURE 6-4: Communication protocol implemented using three pairs of pipes 68

FIGURE 7-1: Time performance for test programs 76
FIGURE 7-2: Memory performance for test programs 78
FIGURE 7-3: Scalability in time and space 79

List of Tables

TABLE 7-1: Total execution time of test programs (in milliseconds) 72
TABLE 7-2: GC time of test programs (in milliseconds) 73
TABLE 7-3: Actual execution time of test programs (in milliseconds) 74
TABLE 7-4: Number of CONS operations performed of test programs 75
TABLE 7-5: Virtual memory size for test programs (in kilobytes) 77
TABLE 7-6: Time and memory consumed using different recording granularities 77
TABLE 8-1: Comparisons among four methods of making data objects persistent 84

CHAPTER 1 Introduction

This thesis describes a technique of *lazy debugging* using persistent data structures. *Lazy debugging* is a new and powerful approach to debugging programs written in high level programming languages. We begin by reviewing previous and current research work related to our subject, go on to describe ideas leading to the development of our incremental debugging model and lazy debugging approach, and discuss a lazy debugger prototype designed and implemented for the SETL programming language. This chapter notes some major problems with current program debugging tools, summarizes the work reported in the remainder of the thesis, and gives an overview of its organization.

1.1 The Problem

Debugging, as defined in the current ANSI/IEEE standard glossary of software engineering[33], is the process that serves “to detect, locate, and correct faults in a computer program”. A fault, generally referred to as a bug, is a condition causing the computer program to fail to perform its required function. Among the three main purposes that debugging may address, systematic detection of the presence of bugs forms a self-contained research area called *program testing*, and is not at issue in our research. We focus our effort on the way that program bugs are located once known to

be present instead, because locating bugs in a program is generally more difficult than correcting them, and because bugs are often easily corrected once they are precisely located[53].

We refer to the computer program to be debugged as the debugging target. Program debugging tools, generally referred to as debuggers, are software systems that help people locate program bugs by making it easier to investigate the execution history of a debugging target. The execution history of a program, also known as the runtime history, refers to the sequence of operations that occur during the program's execution, as defined by the programming language in which the program is written. To debug a program is in part to explore its execution history, proceeding on an ad hoc basis that evolves during the course of debugging itself. The problem on which we will focus is how to cope with the limited accessibility of this history in debugging as ordinarily conducted.

Despite the high cost of program debugging and its known difficulty, debuggers are much less used in software development than might be expected[60]. Several factors contribute to their limited use. Debuggers are complex systems that are difficult to design and implement well. Their success is heavily dependent on their interaction with many underlying system components, e.g., compilers, operating systems and user interfaces. Debugging has traditionally used crude tools such as core dumps and breakpoints, which are highly driven by efficiency considerations. Current debuggers are often difficult to use, and people often have to adjust their debugging style in uncomfortable ways to use them, as in the old Chinese saying: "Cutting off the feet to fit the shoes".

Before summarizing our research, we should point out that quality software depends more on formal design methods than on frequently uses of debugging tools. Debuggers are complement rather than substitute to formal design. We believe that programmers should not resort to debuggers whenever a bug occurs, and that use of debuggers is not an excuse for hacking.

1.2 Summary of Research

Our research aims to provide an improved debugging tool via a new debugging model that supports powerful and high-level forms of program debugging. We aim to create a program debugger that is:

1. Powerful in functionality, in that it
   - Provides a static view of the whole of program execution history,
   - Allows easy, quick and systematic examination of large masses execution information, and
2. Easy to use, in that it
   - Helps users locate bugs without requiring multiple debugging runs,
   - Provides a simple, consistent, and efficient graphical user interface,
   - Promotes a new debugging strategy independent of conventional debugging command languages.

To this end, we introduce an incremental debugging model based on a lazy debugging approach. This provides an alternative to current models and approaches, which are most commonly based on the breakpoint technique.

Our incremental debugging model recognizes that it is very unlikely that people can locate program bugs without doing a lot of data examination. Thus we view debugging as an iterative and incremental process, and accordingly aim to support easy and quick exploration of large amounts of execution history, and ease the detection of relations among data views.

Our lazy debugging approach reflects this incremental view of debugging. It gives debugger users access to a program's full execution history and a powerful set of data browsing and investigation tools. We achieve this by postponing investigation of the internal behavior of a debugging target until its execution is complete and the debugger has recorded its complete execution history. Execution history is then viewed through an easy-to-use graphical user interface. One of the major innovations of this approach is

use of a runtime system implemented by means of persistent data structures, which can record execution history efficiently. Because of recent advances in the methods available for making data structures persistent, the accrued amortized time and space costs for the required information recording are low enough that they are tolerable for debugging of substantial systems.

The post-mortem debugging style and the availability of a program's complete execution history make it possible to build a powerful user interface supporting our incremental debugging model. This helps users locate program bugs naturally and efficiently. A primary design concern in building such a user interface is to make the execution history accessible to users at multiple levels of detail. We have developed several graphical techniques to this end. These include multiple views of execution information, data browsing and traversal using direct manipulation, automatic update, animation of text and graphics, and others. This makes our user interface transparent and also supports new debugging functions, such as forward and backward control breakpoints, and forward and backward data breakpoints. Conventional techniques such as single stepping, memory dumps, and tracing are also easy to support.

To demonstrate our approach, we have implemented a visual debugger prototype for the SETL programming language. This includes a SETL runtime system that can record the complete execution history of a debugging target, an innovative graphical user interface that allows users to browse and examine the recorded information easily and quickly, and a set of debugging routines.

Overall, the primary contributions of our research are:

1. Definition of an incremental debugging model and lazy debugging approach, which together define a powerful paradigm for building highly usable debugging tools.
2. Construction of a SETL debugger prototype, which demonstrates that the lazy debugging approach can be implemented efficiently using persistent data structures.

3. Our SETL debugger prototype also demonstrates the value of a powerful graphical user interface for debugging, and shows how much the incremental debugging style can improve debugger usability.

1.3 Overview of Dissertation

This chapter briefly notes some major problems of current debuggers and outlines the debugging approach we propose.

Chapter 2 discusses general debugging issues, reviews research directly relevant to our work, and describes our new debugging approach.

Chapter 3 contains a detailed debugging example which illustrates key features of the SETL debugger that we have implemented.

Chapter 4 details the design and implementation of a major component of our SETL debugger. It describes the specially designed runtime system used to store multiple runtime states of SETL program executions efficiently. Chapter 5 describes the design and implementation of our debug interface in more detail. Chapter 6 explains the internal structure of the system.

Chapter 7 summarizes the results of our research, which include a performance analysis of the runtime system implemented and a discussion of usability issues of our debugger's interface. Chapter 8 reviews our research in the light of these results. It addresses some of the limitations in our current design and implementation, considers possible improvements, and sums up our research.

CHAPTER 2 Debugging Issues

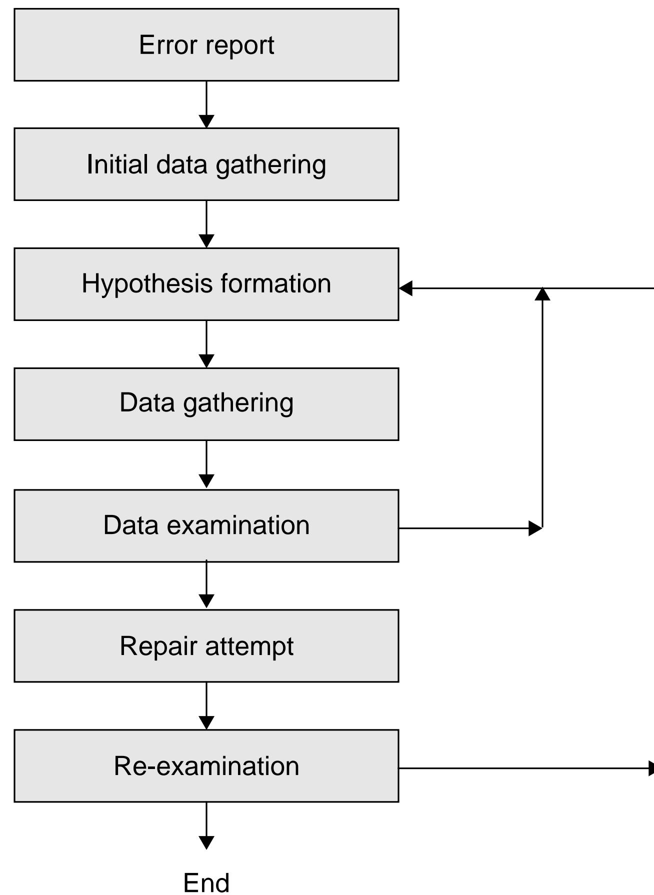
This chapter begins with a general discussion of program debugging issues, goes on to review related work directly influencing our research, and finally describes our debugging model and approach.

2.1 General Debugging Issues

Program debugging has been studied since the introduction of electronic computers and programming. People have since developed various models to abstract the debugging process. These debugging models embody views of the way in which program bugs are located, and hence implicitly influence debugging tools design. Debugging models also affect the way in which debugging tools come to be supported by underlying systems and the way they are used. All this directly affects debugger usability.

Figure 2-1 details a typical current debugging model[8]. Starting with the *error report* and *initial data gathering* stages, this model views program debugging as a process consisting of two loops. The inner loop has three stages: a *hypothesis formation* stage generates hypotheses concerning error locations from the data gathered up to a given point; a *data gathering* stage collects the data necessary to verify the hypotheses formed; and the data gathered are examined in the following *data examination* stage. After

FIGURE 2-1: A detailed debugging model



verifying or refuting a hypothesis concerning the source of a problem, one can continue with new hypotheses verification, or attempt to fix the bugs that have been located, possibly leading to further examinations of the target program.

We can make several remarks on this model. First, it reflects the idea that debugging is an iterative process. It is very hard to locate a bug from an initial error report directly. One must repetitively form hypotheses concerning what have gone wrong and gather additional execution information to verify or refute these hypotheses. Initial investigations often fail to pinpoint bugs; instead they serve to clarify underlying problems, so that following investigations can achieve better focus, reflecting from the

better understanding accumulated. In this sense, debugging is also an incremental process.

The better understanding accumulated as the debugging proceeds gradually reveals the importance of implicit relations and/or constraints among data items and structures used in the debugging target. It progressively reduces the space in which bugs must be sought until it is small enough for one or more bugs to be easily pinpointed.

An important consequence of the above remarks is that debugging is a manual process, likely to defeat all attempts to design the automatic debugging tools that have been studied so far[69][19][27]. A primary concern in designing highly usable debugging tools is to improve the limited accessibility to execution history, which is the most significant drawback of existing systems. In other words, debuggers must aim to provide the most effective possible support for the three middle, most frequently traversed debugging stages of debugging, as depicted in the preceding model.

2.2 Related Research

This section reviews research in four areas related to our work, in the light of the debugging issues discussed above. These areas include design of program debugging, program monitoring, debugging user interfaces, and design of persistent data structures.

2.2.1 Program Debugging

It is well known that current program debugging tools are hard to use well. In the following sections we discuss ways in which the key stages of the debugging process are customarily handled.

2.2.1.1 Print Statements

Inserting print statements is the most primitive way of debugging a program. Nevertheless, this very primitive technique is hard enough to improve upon, and remains in very wide use. For example, Wisenstadt reports that print statements and

hand-simulation are primary debugging techniques used in 80% of the debugging attempts he gathered[24].

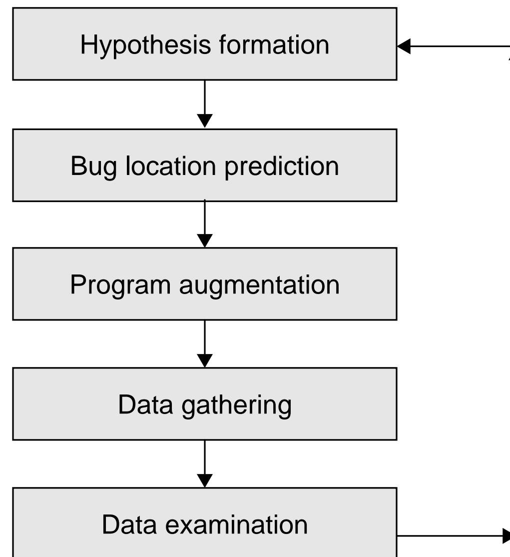
Nevertheless, this elementary technique does not serve debugging very well. Its first and most serious shortcoming lies the high cost it incurs to examine program data, and the relatively fragmentary data it generates[66]. For each error hypothesis formed concerning a debugging target, users have to augment the target program with appropriate print statements and execute this augmented program. It is generally very hard to predict the locations in which print statements should be inserted until users have achieved a fairly good understanding of the internal behavior of their debugging target. This may only be achieved after many unsuccessful debugging executions with different debugging printouts. Another shortcoming is that users need to modify their debugging target to insert debugging print statements and delete or comment them out after debugging, a process that is annoying and may also affect the behavior of the recompiled code. Finally, print statements do not easily support correlation of debugging output results to program source lines. Therefore debugging using print statements generally requires multiple *predict-modify-run-print-examine* cycles to locate a bug, creating server inefficiencies that are well known[66].

Put into the perspective of the debugging model discussed above, debugging using print statements does not support the data gathering stage well. It adds two additional stages to the inner loop of the debugging process, as shown in Figure 2-2.

2.2.1.2 Breakpoint Debuggers

More sophisticated current debuggers, such as dbx[42] and gdb[75], are often *breakpoint debuggers*. These debuggers allow users to halt execution of their debugging target at specific execution points and examine the data objects as they stand at those points. Such an execution point is generally referred to as a *breakpoint*. Breakpointing can be triggered either by execution of a source line or procedure invocation (*control breakpoint*), or by modification of specific program variables (*data breakpoints*). When they correlate breakpoints to source lines and/or variables, breakpoint debuggers are called source-level debuggers.

FIGURE 2-2: A debugging model for use of print statements

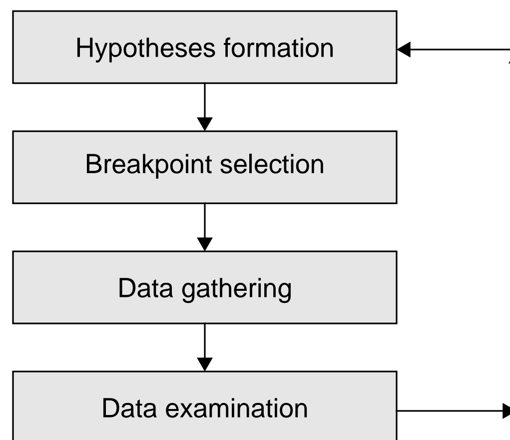


Although they only give intermittent views of a program execution, breakpoint debuggers allow dynamic exploration of execution information. In contrast to simple use of inserted print statements, breakpoint debuggers avoid examination of data objects until an execution reaches a breakpoint. Once at a breakpoint, users can issue as many output commands as desired. This gives users a more flexible way of exploring an hypothesis than simple print statements do. Moreover, the source code of the debugging target remains unchanged when using breakpoint debuggers.

Unfortunately, breakpoint debuggers only respond when an execution stops at a breakpoint. This leaves their users with the task of predicting which execution points are worth examining. Wrong predictions result in breakpoints at which nothing useful is revealed. Even worse, if users miss the right execution cycle for examining program data, they may have to re-execute the debugging target after setting different, possibly earlier, breakpoints; or possibly the right cycle is still to come, in which case they must either step through the execution tediously till reaching the right point, or make another chancy prediction.

For these reasons, debugging with breakpoint debuggers is of limited effectiveness and generally requires multiple debugging sessions to locate a bug. That is, multiple *predict-run-print-examine* cycles are required, as shown in Figure 2-3. However, the technique of postponing data examination until program execution reaches a breakpoint does make breakpoint debuggers considerably more flexible and useful than simple use of print statements.

FIGURE 2-3: A debugging model for breakpoint debugging



2.2.1.3 Reversible Execution and Dynamic Slicing

Both breakpoint debuggers and print statements require multiple executions of a debugging target to locate a bug, and neither of them supports flexible data examination at arbitrary execution cycles. These problems have been recognized in prior research[5][71] and can be characterized as low functionality and high complexity. Various possible improvements have been described[5][80], e.g., reversible execution and dynamic slicing. Both these methods allow execution information to be collected only after it is known to be useful to current hypothesis verification at a given breakpoint. These approaches have direct influenced described in this thesis.

In order to collect prior execution information, reversible execution checkpoints a program execution by recording its execution state periodically. It can therefore roll back

and re-execute a program from an earlier checkpoint to generate necessary information when required. Checkpoint selections and recording of associated data can be done automatically by instrumenting a source program with special event recording routines at crucial execution points, e.g., the beginning of functions or the head of conditional branches. Design of systems of this sort must aim at reducing the re-execution cost by providing enough checkpoints to make the execution interval of any two consecutive checkpoints small enough that any necessary re-execution is small and quick. Although powerful, this approach has often been avoided because numerous execution state checkpoints can consume huge amounts of memory space and significantly degrade system performance[11]. One of the few successful exceptions to this assessment is a debugger developed for the ML programming language[80]. As a functional language, ML[47] has only a few non-functional effects (mutable store and I/O operations). The functional part of ML execution can therefore be captured completely in a continuation[26], a feature supported directly by the language and by the SML-NJ compiler[7]. Continuation recording is much more efficient than raw memory dumping. However, continuation recording has limited applicability, in the sense that only sparse use is possible. Another problem with this method is that it is restricted to functional programming languages and not applicable to imperative programming languages such as Ada, Pascal, C, or SETL.

Debugging using dynamic slicing[4][5][44] employs a different approach to support examination of prior execution information. Instead of checkpointing every important execution point, this method performs a separate step of program dependency analysis, and checkpoints only those points which are directly or indirectly relevant to a given set of breakpoints with respect to a specific execution input. A point is relevant if and only if its execution affects the runtime state of the breakpoints given. These points constitute what is called *dynamic slice*[37], which is "a subset of the statements and control predicates of the program which directly or indirectly affect the values computed at a given [breakpoint] criterion"[79]. Dynamic slicing checkpoints only a portion of execution but still provides complete history for verifying a specific hypothesis during debugging. Unfortunately, the presence of function calls, pointer and procedure aliasing, and even

loops in a debugging target forces dynamic slicing to take numerous checkpoints and degrades performance significantly.

Each of these two approaches instruments a debugging target with checkpoints that record snapshots of its execution state so that users can go back to examine prior information at lower cost than full re-execution would require, even though a small amount of re-execution may be necessary. Still the high overhead of repeated checkpointing severely limits the use of these approaches in practice.

2.2.2 Program Monitoring

Program monitoring[62] usually employs software or hardware packages that view program execution as a sequence of events, which are accumulated as an execution event history. This event history can be analyzed subsequently to a run. Program debuggers based on program monitoring techniques are known as *event-based debuggers*. This style of event-based debugging formerly lay somewhat outside the mainstream of debugging research, but it is getting more popular, especially in concurrent program debugging[46].

Execution event history can be used in various ways:

1. Browsing. Users can browse an execution event history through a text editor or other graphical utility. McDowell and Helmbold summarize four basic techniques that are commonly used to display event history information:
    - Textual and graphical presentation of a snapshot of data objects at an execution moment, which may involve color, highlighting, etc.
    - Time-based diagrams presenting a program history in a two-dimensional display in which time is one axis and some aspect of execution events (e.g., the execution stack level) is shown on the other axis[28][31].
    - Animation by displaying snapshots one after another, to show the dynamic properties of program data objects[16][48].

- Multiple and simultaneous views of program execution information using multiple windows[34].

2. Replay and Simulation. A program execution can be replayed, and at least in some simulated approximation, using the events recorded[40][56].
3. Filtering. Parts of an execution event history can be selected by applying appropriate filtering predicates, allowing users to find useful information more easily in a reduced event space[74][41].

However, event-based software systems have severe limitations. The most important drawbacks are performance related, including excessive memory space consumption and significant execution speed degradation. These limitations result from the efficiency sacrificed by prior technical methods of recording and storing event history. These methods ranges from combining breakpoint techniques with automatic printing and resumption[11][46], copying main memory repeatedly[82], linking to special monitoring routines that are a modified version of system routines[28], and inserting additional monitoring statements in source programs or object code[45]. These methods record execution state explicitly and result in substantial overhead. Performance degradation is increased by the primitiveness of the forms in which event history has been stored, varying from plain text files[11] to relational databases[74] to Prolog programs[41][78].

Partial recording schemes, which record only selected events, are therefore almost always adopted to make event-based debuggers practical. But this causes problems similar to those of breakpoint debuggers. Instead of predicting the data or statements at which to break, users of these event-based systems must predict which events to record. Since the partial histories collected are often incomplete in crucial regards, additional executions of a debugging target then become unavoidable.

2.2.3 User Interfaces for Debugging

The facts that debugging is a challenging human task and that computer software must basically play a supportive role make human-debugger interaction considerations

crucial in debugging. Thus, the user interface that a debugger provides has a great effect its usability.

2.2.3.1 Interaction Style

The term of *interaction style* refers to the way that people are encouraged use a debugger. Despite the obvious significance of this question, debugger interaction style issues have been largely neglected in program debugging research. In consequence, most current debuggers (e.g. dbx and gdb) provide an interface that still employs simple text windows into which debugging commands must be typed. Thus turns the interaction between debuggers and users into a kind of dialogue, in which users type in commands and debuggers execute them and display some output in reply. This kind of dialogue tends to be too specific and too low-level to be ideal, simply because debugging actions are often too subtle to be readily stated formally. Furthermore, use of a text-oriented command language introduces an unnecessary indirection in naming program data, which slows the debugging process.

Most available improvements merely provide favorable command structures and improved command names or abbreviations. Even if up-to-date graphical displays are used to relieve users of typing work, the situation improves only marginally. This is because most current debugging user interfaces are designed on the top of command language based debuggers that already exist and few of them are built from scratch[3][34].

One of the interesting existing works on debugger interaction style issues discusses UPS[13]. Although a conventional breakpoint debugger, UPS provides a graphical user interface that supports a number of unusual features. In particular, it manages its display by making use of a hierarchical structure to hide information within higher-level objects. It also supports an editable workplace for inserting of breakpoints and display of variable values. Another example is the visual debugger of Smalltalk/V[64], which also supports hierarchical view of runtime information based on current window-based user interface techniques.

2.2.3.2 Program Visualization and Animation

Recent advances in computer graphics and visualization/animation techniques are also relevant to the design of debugging interfaces. As observed by Schwartz, textual printout in debugging is often not quite desirable: “the abstract objects basic to the program’s logical performance may not be represented in any very explicit manner in the data printed, making it necessary to reckon back, often very tediously by hand, from this data to the more abstract structures on which the mind’s eye needs to focus”[68]. Proper visual presentation of information often improves access and comprehension because it can be simple, succinct and easy to interpret. Commonly used techniques include use of color, special computer graphics, animation, and facilitates direct user manipulation[73].

Many different approaches have been described. For example, program data objects can be mapped into some graphical primitives such as points or shapes. BALSA[16] allows arrays of data items to be easily mapped into various graphic forms. PROVIDE[48] allows sets of data to be mapped into boxes, bar-charts or pie-charts. KAESTLE[14][15] displays LISP data structures in a window system which supports representation changes and graphical layout rearrangement. VIPS[34][70] displays linked data structures in programs written in Ada or C in a manner supporting both overview and selective displays. It also associates these displays with a debugger.

Program visualization techniques for displaying program execution traces have often been described. (This is particularly useful in understanding the execution of concurrent programs[46].) An execution trace display is usually displayed in a two-dimensional diagram, in which one axis represents time and the other axis represents program entities varying over time (e.g., processes or messages). Two examples are mtdbx[28], which displays events over time and IDD[31], which displays the messages passed between processes.

Program animations are produced by displaying execution snapshots (e.g., visual representations of program data objects or data structures) one after another in time order. Program animations can be viewed during program execution or afterward. In

either case, the advantage of program animation is to highlight potentially revealing aspects of the dynamic change of program objects. But because of the lack of efficient methods for recording program execution, use of program animation for debugging has till now been limited.

2.2.3.3 Hypertext

Research on hypertext has influenced the way in which the debugger presented in this thesis provides access to a potentially large mass of execution historical information. The term "hypertext" refers to the non-sequential organization of information fragments that are connected through non-sequential links[18][57]. Unlike conventional, sequential forms of information such as books or reports, hypertext serves well for information presentation on computers, and is strong in organizing large amounts of information if (see [72]):

1. One deals with a large body of information organized into numerous fragments.
2. The fragments relate to each other.
3. Only a small fraction needs to be examined at any time.

A well-designed hypertext enables users to move freely through the information space, while bringing any information to screen as soon as users request it. Quick response and small navigation overhead keep users' cognitive load low and allow them to concentrate on their higher-level task. Although not fully implemented, Nelson's Xanadu project[55] provides a good example of this approach.

2.2.4 Persistent Data Structures

Work on persistent data structures (see Driscoll, Sarnak, Sleator, and Tarjan[23]) is the last, but perhaps the most critical, research area influencing the design of our debugger. The efficient method of making data structures persistent which is described below plays a crucial role in our effort to build practical and high-level event-based debuggers.

Unlike ordinary data structures, which are *ephemeral* in the sense that only the newest version of the structures are available for use and the old versions are destroyed by any change to the structures, *persistent data structures* allow access to multiple versions of changing data structures. Partially persistent data structures support access operations on all data versions while allowing update operations on the newest version. Totally persistent data structures support both access to and update on all versions of historical data. The following discussion focuses on partially persistent data structures. Hereafter, since these are all we need, they are simply referred to as persistent data structures.

There are three basic methods of making data structures persistent. The most naive one is *checkpointing*, which copies all the data each time any data item is changed. The evident disadvantage of this method is that it requires a lot of memory and is also expensive to perform. Denoting the number of nodes in an ephemeral data structure by m and the number of update operations performed on the structure by u, checkpointing requires $\Omega(m)$ time and space for each update are required. Nevertheless, it is used in many systems (e.g. in [80]) despite its expense.

Instead of storing the whole of a large linked structure when one data item changes, the improved *fat node* method only stores the change[23]. The new value is stored with previous values, which results in development of *fat nodes* holding an arbitrary number of pointer values. This approach takes $O(1)$ to record an update and $O(\log u)$ for an access operation. (Wilson and Moher proposed this kind of an approach to make persistent memory (or *demonic memory* in their terms), which is in effect the fat node method as applied to memory pages[82].)

The technique of Driscoll et al[23] reduces this prohibitive cost drastically. The further improved *node splitting* method described in [23] splits fat nodes for efficient access. It accrues an amortized cost no greater than $O(1)$ both in time and in space for each update and access operation, as long as each node in the original ephemeral data structures has a fixed finite number of predecessors. This restriction results from the consideration that an update in a persistent node may cause cascading update in its predecessors, and only with constant-bounded number of predecessors can this method

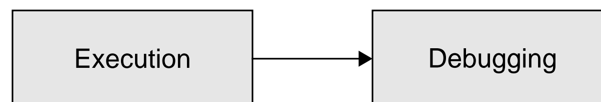
achieve the stated efficiency, as we will see in more detail in Chapter 4. It is this improved method on which the design of our debugger is based.

2.3 Lazy Debugging

Our debugging approach is a natural extension of the debugging approach based on program monitoring. It differs from current event-based techniques mostly in terms of the way execution events are recorded and stored. Our approach uses efficient persistent data structures to record execution events and keep them in memory. We call this *lazy debugging*, because its main idea is to postpone investigation of debugging hypotheses until complete execution information of a debugging target is available to a debugger. Given complete execution histories, users can then debug a program without requiring multiple executions.

We call a debugger built upon this approach a *lazy debugger*. Such systems first record the complete execution history of a debugging target, and then allow users to examine the recorded information in a post-mortem fashion. These two steps can be viewed as *execution videotaping* and *execution review*, respectively. During execution videotaping, execution is divided into timeslices of runtime state and recorded accordingly. Each runtime state recorded is therefore associated with a timestamp. These runtime states together form a *persistent core dump*.

FIGURE 2-4: Lazy debugging separates debugging from program execution



One of the benefits of our lazy debugging approach is that it separates debugging actions from program execution, as shown in Figure 2-4. As we have seen from Figure 2-1, a central debugging cycle consists of three stages: hypothesis formation, data gathering, and hypothesis verification. In lazy debugging systems, these steps can be performed

wholly in the debugging stage, simply because of the availability of complete execution histories.

Our debugging approach also encourages a systematic approach to debugging. Generally, as long as a debugging target is well designed (i.e., conforming to structured or object-oriented design principles), it can be partitioned reasonably into several data or functional components connected by shared data objects. Program execution can thus be viewed as a sequence of component executions. Originally, the bug search space is this whole sequence. Examination of any of the data objects through which these components communicated is crucial. By examining such a data object, we can divide the bug search space in half, depending on whether an error is seen in the data object or not. The second half can be discarded if an error occurs in the object, or the first half otherwise. We can then iteratively partition the reduced space, and eventually locate program bugs by repeated narrowing of focus and data examination.

Even though the lazy debugging approach has great potential for building debugging tools, the debugging process does not necessarily become easier for its end-users unless complete execution histories can easily be examined by them. Just like the printout from a conventional use of debugging print statements, the execution history accumulated by a lazy debugger will generally be voluminous. In this regard, our lazy debugging approach is subject to the objection that the information it accumulates can be larger by orders of magnitude than that produced by the conventional “print-statement” approach.

To deal with this issue, we have developed ways of making historical execution information easy to use, under what we call the *incremental debugging model*. This model organizes execution historical information using structures, which allows users to explore the information organized incrementally, and supports step by step data examination from general information (e.g., execution traces) to more specific information (e.g., variable values at given execution moments). The data examination tools we provide have several aspects:

1. They ease specification of the data object to be examined,
2. They make it easy to specify the execution moment at which this data object is to be examined,
3. They respond promptly to arbitrary data examination requests by users, and
4. They help users detect data constraint violations.

This reduces user data examination expense, allowing them to concentrate more on high-level debugging issues.

CHAPTER 3 A LSD Debugging Example

We have designed and implemented a system called LSD (Lazy SETL Debugger), which is a debugger prototype for the SETL programming language[67], to demonstrate our approach. This chapter presents a debugging example to show various features of this system. Although we give a fairly detailed description of the way our debugger works, a much easier and better way of understanding the system is through a live demonstration. We ask the readers of this chapter who are unable to examine the software to imagine the dynamic graphics employed from the snapshots given.

3.1 Using the Debugger

To work with our debugger, users must employ a sequence of debugging actions, which differ significantly from the actions employed in using conventional debugging approaches. LSD debugging normally employs three explicit steps:

1. Load a debugging target,
2. Execute the debugging target in its special runtime system, and
3. Locate bugs by exploring the recorded execution history using LSD's graphical user interface.

3.2 The Main Window

FIGURE 3-1: The main window

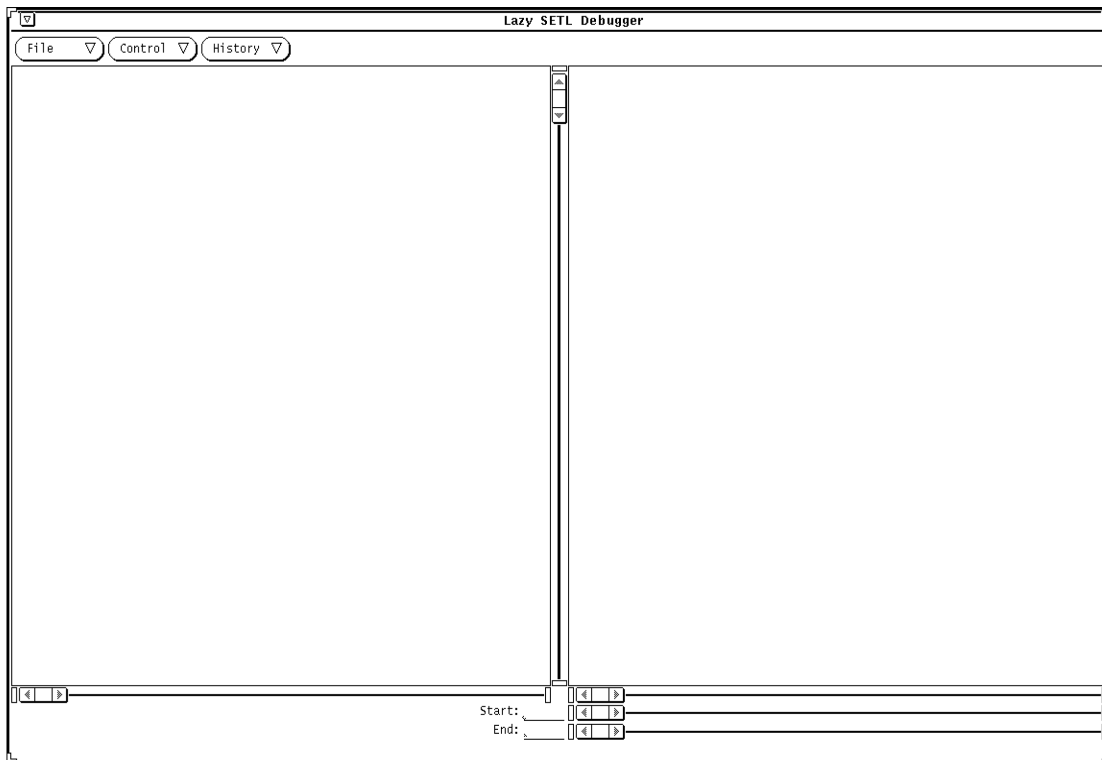
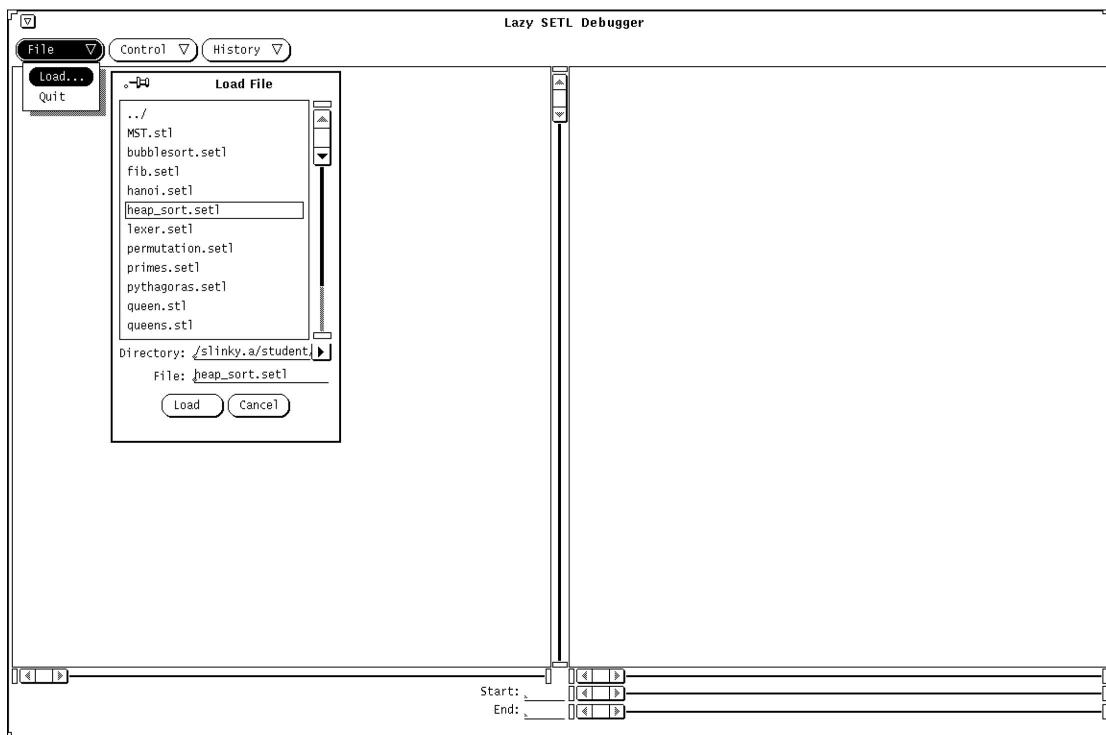


Figure 3-1 shows the main window of our debugger after it is first started. On the left is the source code window, which will show the source code after a SETL debugging target is loaded. On the right is the history window, which will show the execution trace of the debugging target after its execution is complete. Users can adjust the two windows by manipulating associated scrollbars. At the top of the main window there appears a list of menus that controls the primary actions of our debugger, including program load, execution, and execution history update.

3.3 Loading a Debugging Target

Once the main window appears, users can load a debugging target by popping up a load-file window and selecting the Load menu button in its File menu, as shown in Figure 3-2. The load-file window has two parts: at the top is a scrolling list showing the directories

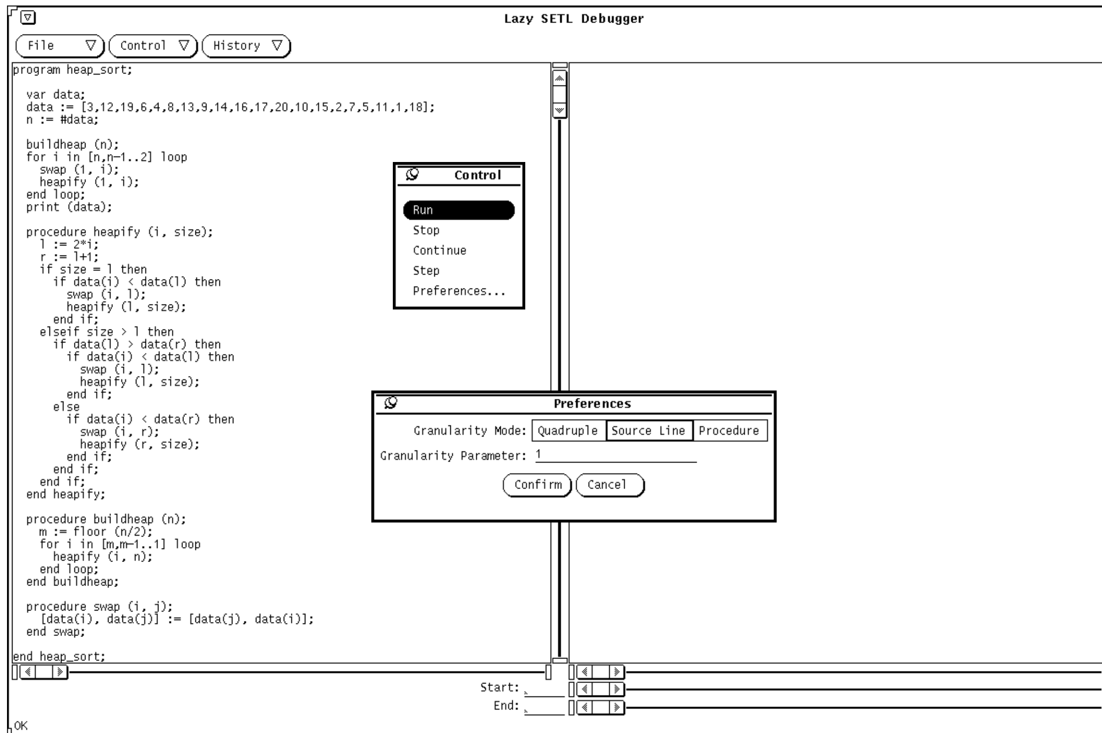
FIGURE 3-2: Load file



and SETL files that are available in the current working directory; at the bottom are two text fields specifying the current working directory and the SETL file selected, and two buttons for loading the selected file (the Load button) or canceling the load file action (the Cancel button). In this example, we load file heap_sort.setl, a buggy heap-sort program that should, but does not successfully, sort a tuple of integers into non-descending order. The source code is seen in the source code window, once loaded (see Figure 3-3). For brevity, we have omitted program comments.

3.4 Program Execution

FIGURE 3-3: Run program



To debug a program, users of LSD must run the program first, and then come back to examine the source code and its execution history in a post-mortem debugging style.

Our debugger has three options controlling the granularity of execution history recording. They are chosen by popping up a preferences window from the Preferences menu button of the Control menu, as shown in Figure 3-3. In this example, we select the default single line granularity.

Figure 3-3 also shows the other four menu buttons of the Control menu, which allow users to control execution by stepping through the following four run modes:

1. *Before run*. This is the initial execution mode after a debugging target is loaded. Users can start executing the program by clicking a Run or a Step menu button. Clicking

Run starts execution and enters the *running* mode (see below), while clicking Step starts an execution and stops it after its first execution step is complete. The size of execution step varies depending on the selected recording granularity.

2. *Running*. This is the normal execution mode during which a debugging target runs in the persistent runtime system until it stops or after a Stop menu button is pressed. If execution finishes normally or on encountering a runtime error, it enters the *after run* mode. Otherwise, a Stop button has been pressed and the execution enters the *run stopped* mode. When an execution stops, its execution trace up to the last execution moment is shown in the history window.
3. *Run stopped*. An execution stopped by Stop enters this mode. A stopped execution can always be resumed using a Continue or a Step buttons, which are self explanatory. The *run stopped* mode is the same as the *after run* mode in other aspects.
4. *After run*. A completed execution reaches this mode. Its execution trace is displayed in the history window. This mode is the normal starting point for debugging.

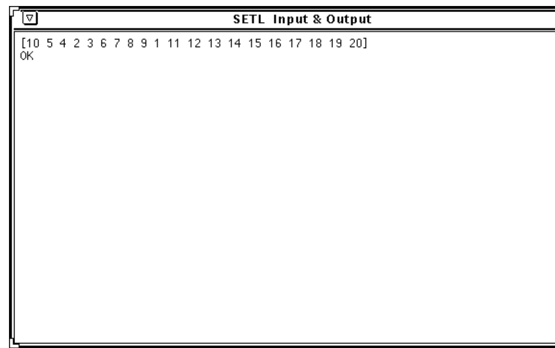
3.5 The Input and Output Streams

Once execution starts, our debugger pops up a separate window that simulates the program's input and output streams (see Figure 3-4). This window reads user input, displays output, and prints out any runtime error messages generated, or prints "OK" if it terminates normally. Figure 3-4 shows a normal termination. Plainly, however, the defective heapsort program chosen for the present illustration did not sort those numbers correctly.

3.6 The History Window

Figure 3-5 shows the execution trace displayed in the history window after execution is complete. The horizontal axis of the history window represents execution time, which is divided into many slices (*timeslices*) by different execution events, while its vertical axis represents program source lines. This window shows the complete trace information of

FIGURE 3-4: The input and output streams, showing a normal program termination

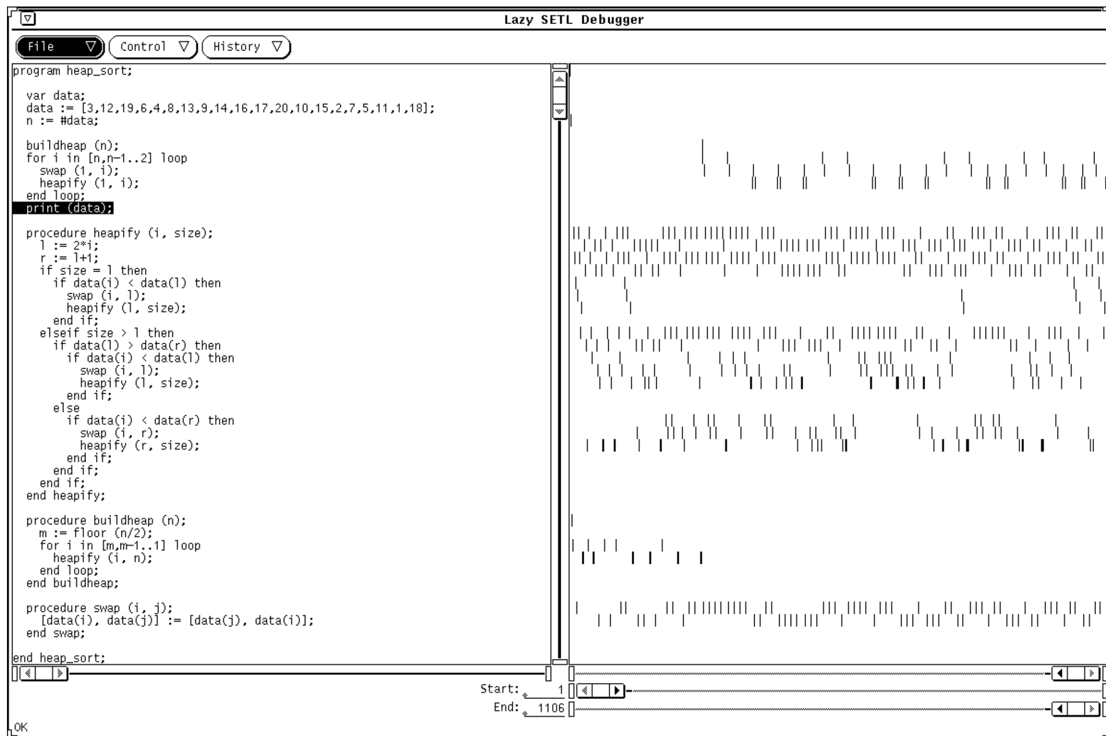


program execution: a mark is placed at location (*x*, *y*) if the *y*th line of the program source code is executed at the *x*th timeslice. For example, it is easy to see from Figure 3-5 that procedure heapify and procedure swap were activated throughout the execution, procedure buildheap was only touched at the beginning, and that a loop occurred in the main procedure after buildheap was complete.

Because of screen space limitations, the execution trace shown in the history window can not usually be complete and detailed at the same time. The history window is about 500 pixels wide while a program execution trace usually has more than 500 timeslices. For example, the execution trace in the current example has 1106 timeslices, which means that less than half of the execution timeslices can be shown in the history window, even if the mark shown for each timeslice is only one pixel wide.

Users can adjust their view of the history window by dragging the three scrollbars located below it. The second and third scrollbars are called the *starting bar* and *ending bar*, respectively, and each is associated with a text field specifying the timeslice to which it points. The timeslice specified by the starting bar is always no later in time than that of the ending bar, so that the two bars together specify a portion of the full execution trace that can be shown in the history window. Initially, the starting bar points to the first timeslice and the ending bar to the last timeslice, so that the whole execution trace is shown. Figure 3-6 shows that dragging the two bars closer effects a zoom along the temporal dimension, i.e., shows a smaller portion of the history in a more detailed view.

FIGURE 3-5: The history window, showing an execution history

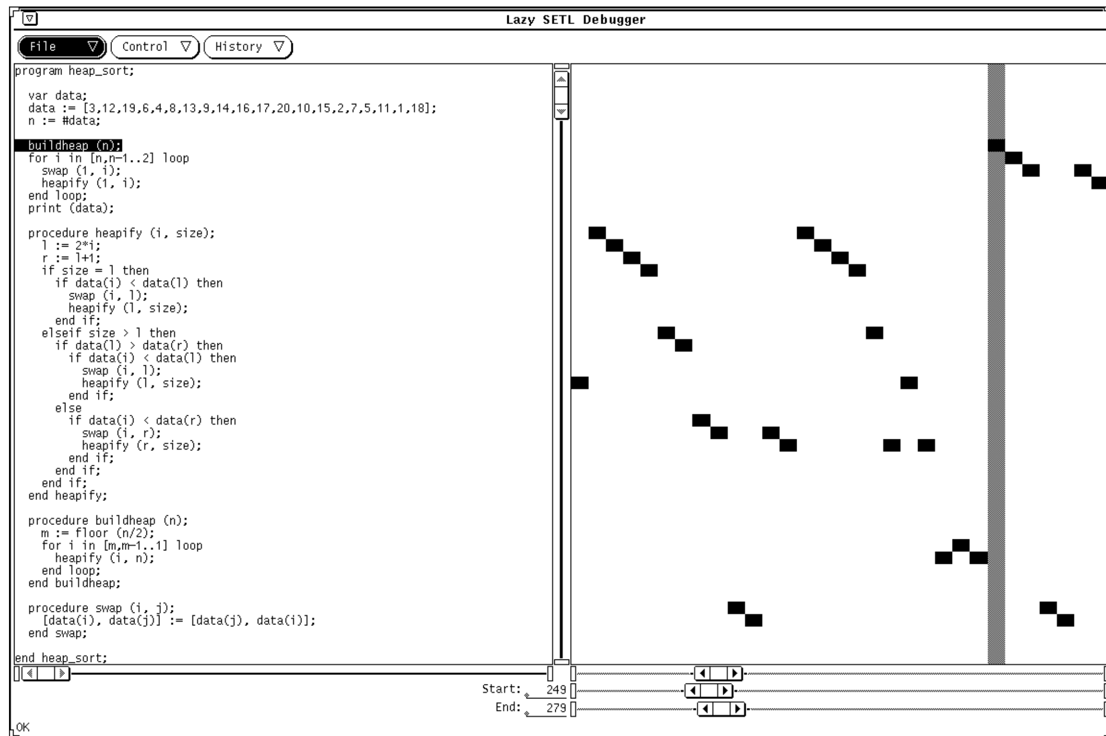


The first one of these three scrollbars is called the *focus bar*, since it specifies the timeslice of current interest. This timeslice is called the *current timeslice*, and is always displayed and highlighted in the history window. The source code window then highlights the corresponding program source line. In Figure 3-6, for example, the focus bar points to the source line buildheap (n); and the history window shows a small portion of the execution trace (31 timeslices) close to the execution of this source line.

3.7 The Stack Window

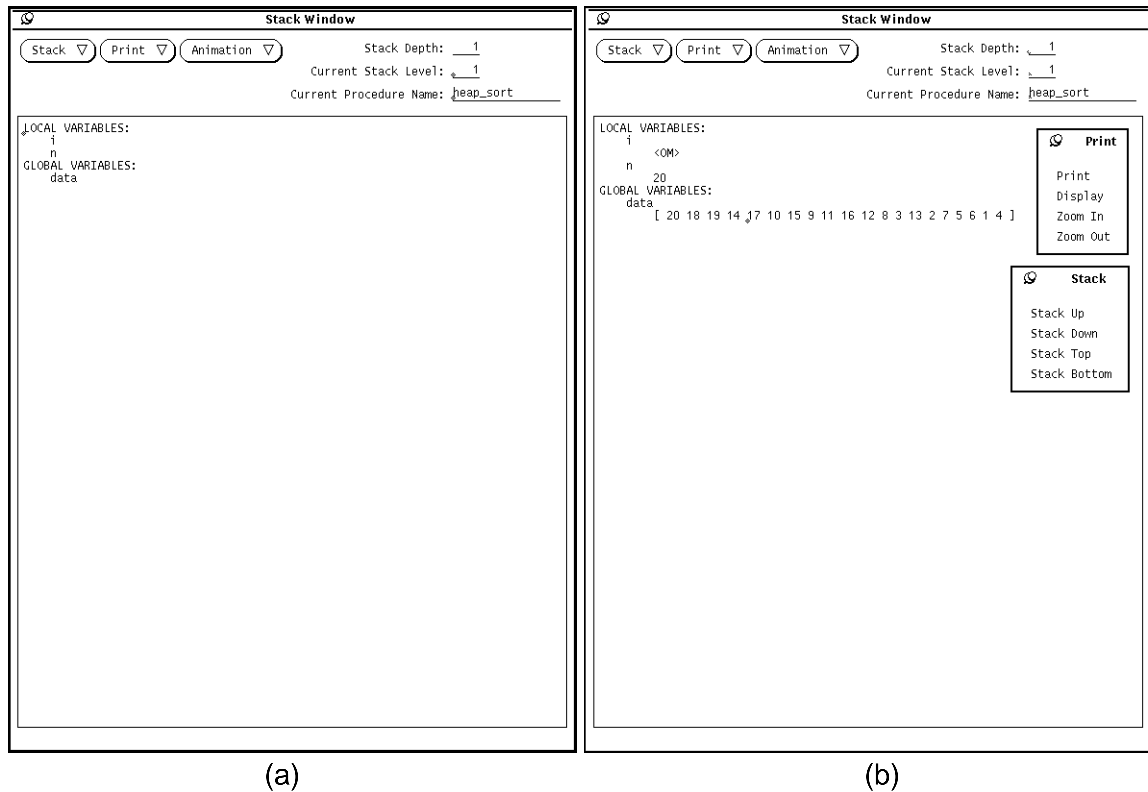
To examine more detailed execution information available at the current timeslice, users can pop up the stack window, by selecting the Popup Stack Window menu button from the history window's popup menu, or by double-clicking the timeslice. As shown in part (a) of Figure 3-7, the stack window has two parts: at the top is a list of menus and

FIGURE 3-6: Zoom effect in the history window



three text fields specifying the depth of the execution stack, the stack level shown, and the procedure name of the stack frame being examined. The bottom of the stack window is a display area showing a list of program variables that are accessible in the current timeslice, including both the local variables on the current stack frame and the global program variables. Using the Stack menu, users can walk through the execution stack, examining arbitrary stack frames. Following the convention of compiler and debugger design, the execution stack in LSD increments downward, and so does its stack pointer. Initially our debugger selects the innermost stack frame pointed to by the stack pointer. Users can select the enclosing frame selecting the Up menu button, the enclosed frame selecting the Down button, the outermost frame selecting the Top menu button, and the innermost frame selecting the Bottom menu button. Changing the stack frame displayed automatically updates the list of accessible variables shown.

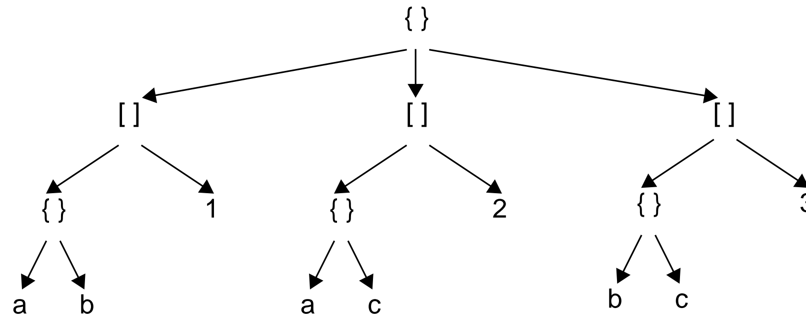
FIGURE 3-7: The stack windows, showing program variables



3.8 Printing Variable Values

To print the value of a variable listed in the stack window, users need to select its name and a printing mode from the Print menu. Our debugger provides three printing modes: Print prints the value of a variable; Display shows the same value as a tree-like structure with proper indents and thus produces a attractive print; Zoom In can show the value step-by-step from the beginning to the complete result obtained from Display. Unlike Print or Display, which treat a SETL variable as a single object, Zoom In views the variable as a value-tree, in which a simple value is a leaf node and a compound structure is an internal node, as shown by an example in Figure 3-8. When the Zoom In printing mode is applied to an internal node of a value, the sub-value tree rooted by the internal node is

FIGURE 3-8: SETL values are treated as trees in printing

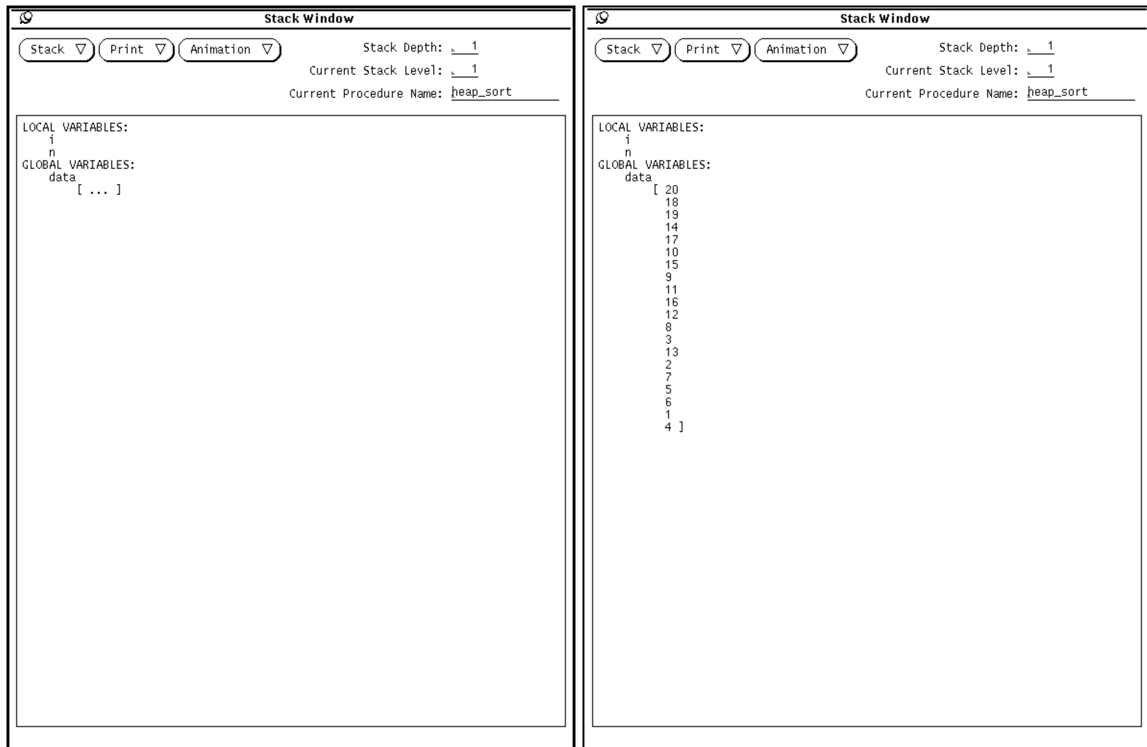


Value tree for SETL value {[{a b} 1] [{a c} 2] [{b c} 3]}

elided using a ***zoom mark*** (…). For example, an elided non-empty set is displayed as "{…}", and an elided non-empty tuple is displayed as "[…]". A further printing on an elided value is applied to each of its sub-values. Therefore, Zoom In allows users to examine a complex value interactively and incrementally, and is naturally called ***interactive displaying***. Finally, printing operations by Print, Display and Zoom In are always reversible by Zoom Out. Part (b) of Figure 3-7 shows, in simple Print form, the values of our debugging target's variables at the execution moment when the procedure call buildheap (n) has just returned.

Zoom In printing mode is a very useful way of printing complex SETL values. SETL allows two kinds of data values: simple data values (integer, real, string, boolean and atom) and compound data values (tuples and sets). SETL compound values are very flexible and can have elements which are themselves (simple and compound) values. This can result in very complex values that are hard to examine using conventional print facilities. Two typical examples are compiler parse trees and symbol tables. Using the interactive displaying, users do not have to print a complex value all at once. Instead they can start data examination from top-level data and explore any further details of values if necessary. Even though in our heap-sort example there is no very complex data object, we can still get the flavor of interactive displaying by applying Zoom In to the tuple-valued variable data. Figure 3-9 shows the results: Part (a) of Figure 3-9 shows the result when

FIGURE 3-9: Interactive displaying



(a) (b)

applying Zoom In to the variable data, indicating that it is a tuple; Part (b) of Figure 3-9 shows the result by applying Zoom In to the zoomed value in Part (a), giving a final result as it reaches the deepest level of the value tree. A more complete illustration is shown in Figure 3-10, using a SETL variable graph = {[{a b} 1] [{a c} 2] [{b c} 3]}, which is not available from the current debugging target.

3.9 Program Animation

As we have seen so far that our interface has multiple windows. A crucial property of these windows is that they are well coordinated in the interface. The key aspect of window coordination is the commonality of execution time, as represented by the notion of a current timeslice. When an update in the current timeslice occurs in one window,

FIGURE 3-10: Examples of variable printing

Print

```
graph
    {[{a b} 1] [{a c} 2] [{b c} 3]}
```

Display

```
graph
    {[{a
       b}
      1]
     [{a
       c}
      2]
     [{b
       c}
      3]}
```

Zoom In

```
graph
    {...}
```

Zoom In + Print

```
graph
    {[{a b} 1]
     [{a c} 2]
     [{b c} 3]}
```

Zoom In +Display

```
graph
    {[{a
       b}
      1]
     [{a
       c}
      2]
     [{b
       c}
      3]}
```

Zoom in + Zoom In

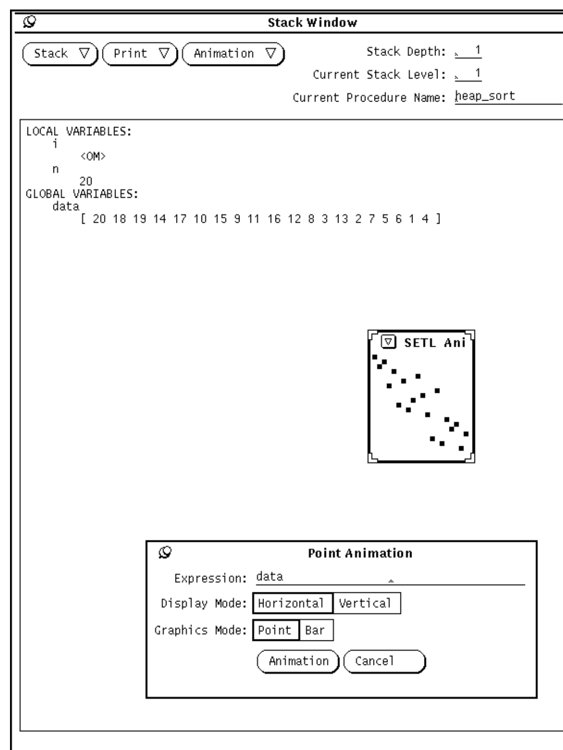
```
graph
    {[...]
     [...]
     [...]}
```

other related windows are updated simultaneously and automatically. Among other things, this coordination naturally supports program animation.

To animate values of variables, for example, users can change the current timeslice (e.g., by dragging the focus bar) while showing these variables in the stack window. Printing generates snapshots while the value changes with the current timeslice. Program animations are also available from the source code window and history window. Similarly, users can show the animation of source code execution in the source code window, which highlights current source lines at each execution moment in the context of the source program; while they can also show a similar animation in the history window, which highlights current trace marks in the context of program execution trace history.

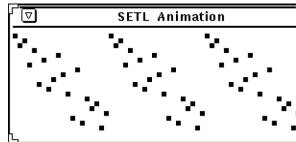
Other graphical forms are also available for some SETL values that can be mapped directly into graphical primitives such as points or lines. In this debugging example, we can map the tuple-valued variable data into a set of points, where a point is a tuple of two integers (an integer item and its index in the tuple)[16]. Figure 3-11 shows a snapshot of this graphical representation.

FIGURE 3-11: Animation snapshot



One can also animate any SETL expressions having appropriate graphical representations. For example, users can animate the expression data*3 instead of the simpler data, if there is any reason to do so (see Figure 3-12). This is achieved by an internal animation routine that can evaluate SETL expressions during debugging. The implementation of this will be discussed in more details in Chapter 6.

FIGURE 3-12: Animation of the SETL expression data*3



Users control program animations by changing the current time slice, from one of three windows in the debugging interface: selecting source lines in the source code window, selecting trace marks or manipulating the focus bar in the history window, or clicking variable names in the stack window. and having some variables printed, users can also animate the change of these variable values. They can also control animation speed and direction. However, the animations concerning variables values are restricted by a *same scoping* restriction, which requires that the animations are performed in a same scoping, and therefore on same variables. This assures that program animations are coherent and meaningful in the interface. The textual variable animation is also restricted such that the animation can only use the top-level printing mode, because SETL is a dynamic typed language and lower level printing modes may become obsolete due to an assignment.

3.10 Simulation of Conventional Debugging Facilities

LSD can simulate many conventional debugging facilities and also support a number of new ones which are usually not available or difficult to implement in conventional debuggers. Users can click the forward button (i.e., the right arrow) of the focus bar to go to the following timeslice, which is forward stepping. They can also click the back arrow to go to the preceding one, giving backward stepping. Clicking the forward mouse button (i.e., the right button) on a source line in the source code window goes to the next timeslice in which this line is executed (forward control breakpoint); clicking the backward button goes to the previous one (backward control breakpoint). Similarly, clicking the forward mouse button on a variable name in the stack window goes to the

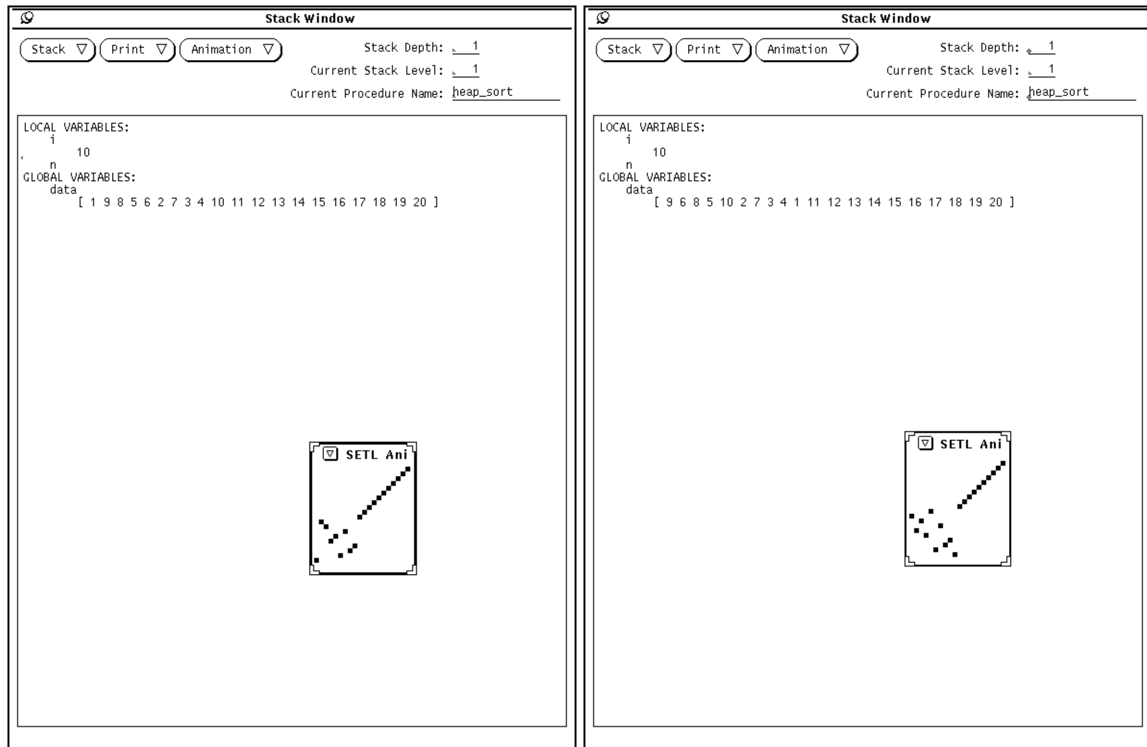
next timeslice in which this variable has a different value (forward data breakpoint); while clicking the backward button goes to the previous one (backward data breakpoint).

3.11 Debugging Example, Continued

We now go on to debug our example program. We know that it has a bug, since the program did not sort correctly. We know that the bug is located at some program point in which a correct execution state is immediately followed by an incorrect execution state. Using the debugger, we systematically narrow down the search space until we have pinpointed statements in which a bug can be easily recognized.

Recall that the heap-sort algorithm sorts a tuple of numbers by first building a heap on them, then swapping the maximum element into its correct position at the end of the tuple, restoring a smaller heap, and repeating these steps until all the numbers are sorted. The first timeslice we want examine is naturally that in which the heap is first built. Part (b) of Figure 3-7 shows the variable values at this moment: variable i was not initialized and had the value <OM>; variable n was 20, which was the size of variable data; data were indeed organized as a heap, which is easiest to see in the animation window. Execution was correct up to this execution point. We therefore proceed to the last part of the execution, which consists of a loop. We can either jump into the middle iteration or step through them, as there are only 20 iterations, to locate the first iteration at which the program "has run off the rails". With the help of the stack window and animation window shown in Figure 3-13, we find that the heap was not restored correctly by the heapify (1, i) statement in the 10th iteration. Knowing this, we go back and step through the procedure call heapify(1, i), with i = 10, and eventually find out that data item 10 was unexpectedly swapped with item 1. After reexamining the parameters passed to this procedure, we find that the heap size parameter was off by 1 and it should have been heapify(1, i-1). After this correction, this program sorts correctly.

FIGURE 3-13: Locating a bug



before the error occurs after the error occurs

3.12 Condensed Execution Histories

Although not illustrated by in this small example program, memory consumption can be a serious problem when debugging a long-run program using LSD, due to the amounts of execution history that are recorded.

Our debugger allows refinement of coarse execution histories to support debugging of long-run programs, whose execution histories can not be recorded completely due to memory restrictions. This allows users to record a coarse-grained history first and refine any interesting part later.Figure 3-14 shows a part of the execution history produced by a coarse-grained recording (runtime state was recorded in every 10 source lines). If users

have located an interesting portion of the execution history (shown in the history window) and need more detailed information, they can focus down to the execution history selected by clicking the Collapse menu button from the History menu. This causes LSD to discard the unselected history portion and expand the interesting part, by re-executing it with a new, finer-grained recording granularity. Figure 3-15 shows the result of such an refinement in which the single source line recording granularity is used. Comparing to the 52 timeslices over this execution period (see Figure 3-14), we now have 520 timeslices (see Figure 3-15), which provide much more detail.

FIGURE 3-14: A portion of a coarse execution history

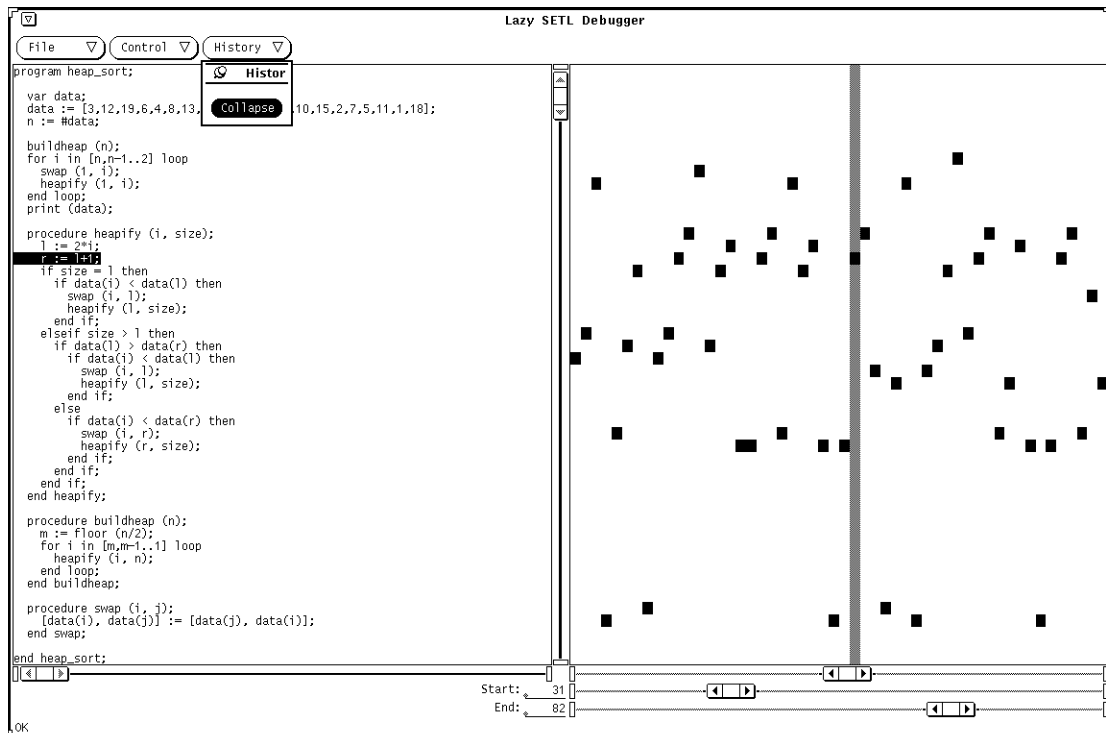
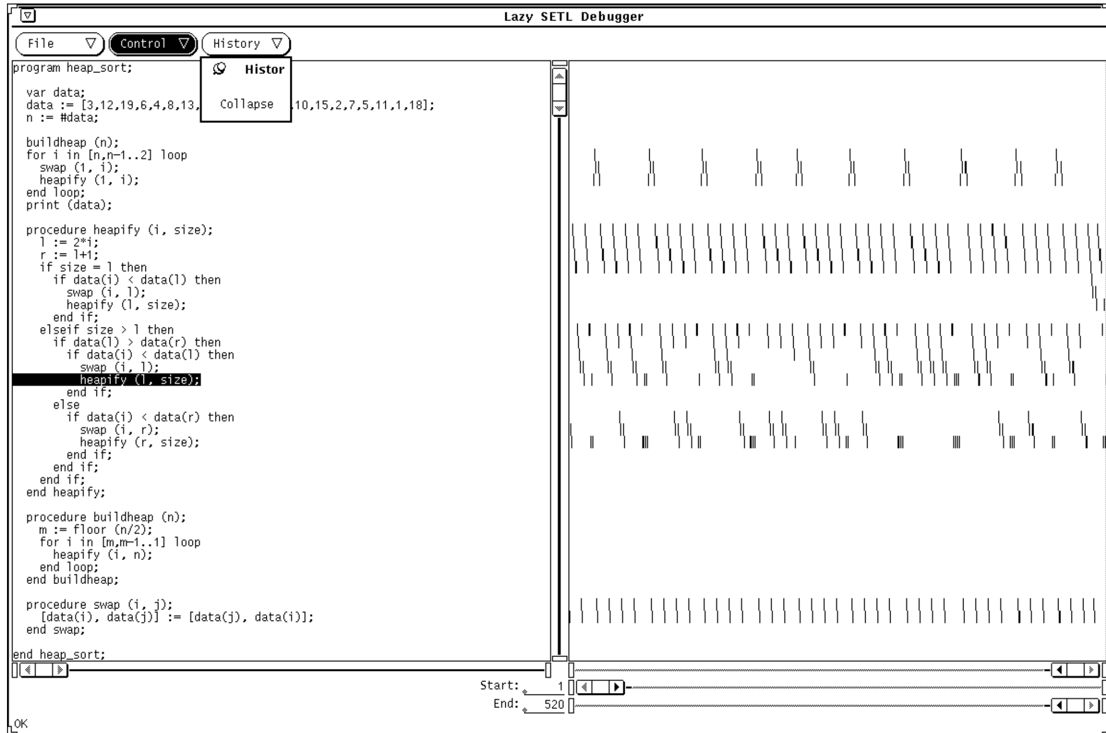


FIGURE 3-15: Refining a coarse history



CHAPTER 4 Persistent Runtime System

This chapter describes the design and implementation of the persistent runtime system for the SETL programming language, which provides the execution historical information used by our debugger. We begin by describing general ideas of persistent runtime systems, discussing the complexity issues and key assumptions. We then go on to describe design considerations for the system implemented and to detail the implementation techniques used, with an emphasis on the persistent data representations.

4.1 General Ideas of Persistent Runtime Systems

As we have seen from the preceding chapter, use of a persistent runtime system is key to the implementation of our Lazy SETL Debugger. A *persistent runtime system* supports entirely normal execution of valid programs written in the programming language for which the system is designed. However, unlike an ordinary runtime system (*ephemeral runtime system*), which keeps only its most recent runtime state, and destroys old states as execution proceeds, a persistent runtime system keeps multiple runtime states so that execution history can be accessed rapidly even after execution is complete. The runtime state record kept gives a complete account of program execution history.

A program execution history recorded by a persistent runtime system includes each piece of subsequently visible information generated during program execution. Note that different programming languages will require somewhat different kinds of execution histories, due to their different execution models. For example, the execution model of imperative languages such as Ada, Pascal and SETL employs explicitly an *execution state*, updated by successive assignment statements. The execution history for a program written in such an imperative language must include control flow information, variable value update and execution stack updates along with procedure activations and returns. (Finer, implementation-dependent details can be omitted, as they are not defined by the language and users should not rely on these details to debug their programs. Such details include, for example, heap management and garbage collection methods.) Similar notions of execution history also apply to functional languages such as ML or Scheme and logic languages such as Prolog.

An important property of a persistent runtime system is that an execution history it records should be complete, in the sense that the history is sufficient enough that execution can backtrack to any previous moment and re-execute, even after initial execution is complete. This property normally guarantees that enough information is available for debugging so that no re-execution is necessary, and allows an execution history to be refined if it is incomplete. As we will see shortly, this property improves debugger scalability.

4.1.1 Complexities of Persistent Runtime Systems

One might fear that use of a persistent runtime systems might result in overwhelming program execution time and memory usage overhead. However, this overhead can be made manageable by using *node splitting* persistent data structures.

To enlarge upon this remark, we introduce some notations. We denote by ERS an ephemeral runtime system, and by PRS a corresponding persistent runtime system implemented using the corresponding persistent visions of the data structures used in ERS. We denote by T_E the time cost that ERS incurs to run a program A, by T_P the time

cost incurred by PRS, by T_o the time cost overhead incurred by one update in PRS, by T_O the total time cost overhead in PRS, by S_E the space cost incurred by ERS, by S_P the space cost incurred by PRS, by S_o the space cost overhead incurred by one update in PRS, by S_O the total space cost overhead of PRS, and by m the number of update operations performed during an execution of program A. All of the above notations are of the worst case.

4.1.1.1 Time Complexity

For any no-nonsense program, it is reasonable to assume that its execution time is asymptotically proportional to the number of update operations performed in its execution, i.e., $T_E = \Theta(m)$. We already know that $T_o = \Theta(1)$. It is then easily seen that T_P is also asymptotically proportional to m:

$$T_P = T_O + T_E = \sum_m T_o + \Theta(m) = \sum_m \Theta(1) + \Theta(m) = \Theta(m) \qquad \text{(Eq 4.1)}$$

which yields:

$$\frac{T_P}{T_E} = \frac{\Theta(m)}{\Theta(m)} = \Theta(1) \qquad \text{(Eq 4.2)}$$

Thus we can expect the time cost of program execution in a persistent runtime system is in the same asymptotic order as that of a similar ephemeral runtime system.

4.1.1.2 Space Complexity

The cost of memory space usage does not behave as favorably as the execution time cost. However, the memory space used by a program execution has an asymptotic order no more than that of the number of update operations performed during the execution, i.e., $S_E = O(m)$. Since $S_o = \Theta(1)$, this gives:

$$S_P = \Theta(S_O) = \Theta\left(\sum_m S_o\right) = \Theta\left(\sum_m \Theta(1)\right) = \Theta(\Theta(m)) = \Theta(m) \qquad \text{(Eq 4.3)}$$

which yields:

$$\frac{S_P}{S_E} = \frac{\Theta(m)}{O(m)} = O(m) \qquad \text{(Eq 4.4)}$$

Plainly, the space cost behavior of a persistent runtime system are not as favorable as the time cost considerations. The increase of the memory usage of PRS over that of ERS can be in the same asymptotic order as program execution time, depending on the actual value of S_E,. In an extreme case, a persistent runtime system will require much more ($\Theta(m)$) memory space if its corresponding ephemeral runtime system can keep on re-using a constant amount of memory (i.e., $S_E = \Theta(1)$).

4.1.2 Assumptions

We have assumed in the foregoing that the data objects used in a runtime system can be organized effectively using data structures which allow the efficient *node splitting* method to be used to attain persistency. This is the case for most modern programming languages, specifically which represent their runtime objects using linked structures, e.g., structures in which local data objects can be accessed from a local environment (or a local activation record) and other data objects can be accessed from enclosing environments which themselves are accessible either directly or from the local environment. However, use of the *node splitting* method requires two additional restrictions, which limit the number of programming languages to which our approach is directly applicable. First, the *node splitting* method does not directly support randomly accessed data objects such as arrays: use of persistent data structures introduces logarithmic overhead for array accesses. Another restriction is that each node in the data structures to be made persistent must have only a fixed number of access points, which implies that pointer aliasing cannot be easily supported. However, the logarithmic overhead incurred to bridge these gaps is generally tolerable. (We will revisit this issue in Chapter 8.)

However, our approach to building persistent runtime systems is almost perfect for a number of programming languages, notably SETL[67] and Icon[29][30]. These languages can be generally characterized as supporting very high-level flexible expressions and

having value semantics. High-level expression implies that randomly accessed memory is seldom required; while value semantics avoids pointer aliasing. As the target language of our debugger, the SETL programming language is our concern. SETL runtime data objects are usually organized using trees, hash tables, and Patricia trees in SETL's two most efficient current implementations. Therefore, SETL persistent runtime systems using the *node splitting* method are readily implemented in a manner guaranteeing favorable performance.

4.2 System Design

4.2.1 Design Issues

This section describes the design of the persistent runtime system of LSD. As persistent runtime systems require more execution time and more memory space than ephemeral runtime systems, keeping the overhead of low is our primary objective. Consequently, we face two key trade-offs in the system design:

1. The trade-off between time overhead and space overhead, and
2. The trade-off between time/space overhead and the completeness of an execution information recorded.

Our underlying design principles are: reducing space overhead has priority over reducing time overhead, and keeping execution histories complete has priority over reducing system overhead. The justification of these two principles is that space overhead is usually more significant than time overhead ($O(t)$ versus $O(1)$), while keeping a complete execution history is crucial to easy debugging. The following discussion focuses on these key design issues, and on related functionality, scalability, transparency, and performance concerns.

4.2.1.1 Functionality

The SETL persistent runtime system is designed with two functional requirements in mind. We must support full, normal execution of SETL programs, while efficiently recording prior runtime states as execution proceeds.

The primary execution information that our system records includes a trace of every source line executed and every update to program variables and execution stack. Some other information (e.g., information connected with SETL iterator expressions such as for i in {1..100} loop) is also recorded in order to support full backtracking of the runtime states recorded. Basically, we design each update operation of SETL execution from its initial overwrite form to the corresponding persistent form. This persistency transformation apply both to execution stack and to the heap in which the values of program variables are stored.

4.2.1.2 Scalability

A second key design issue is scalability. Although our lazy debugging system has moderate time overhead, its memory space requirements can be proportional to program execution time, and therefore unbounded. Since the information accumulated in memory is always historical and does not actually affect current execution, it can always be moved into larger but slower secondary memory. Thus memory consumption is not critical from a theoretical point of view. Nevertheless, it is still an important practical issue, because truly unbounded memory usage must eventually become intolerable. Without control of memory usage, a persistent runtime system that is successful for 2-second programs can fail miserably for a 2-hour one.

The concept of recording granularity is therefore important in our persistent runtime system, which describes a trade-off between the accuracy of the execution history recorded and the amount of memory space required. The most accurate execution history would keep the full runtime states at each source line executed. The less accurate a execution history recorded, the less memory space and time are required. Therefore, a persistent runtime system can trade execution history accuracy for better time and memory space performance.

The recording granularities available in our persistent runtime system are defined by two parameters: the number of runtime states recorded and their distributions. We have provided three basic recording granularities in the current design:

1. Quadruple recording. This, the finest-grained recording granularity, records execution history for each quadruple execution. This level of recording tends to accumulate more information than is defined by the externals of a programming language, and thus sacrifices system efficiency. On the other hand, it provides intermediate data not available otherwise.
2. Source line recording. This is the recording granularity that is normally required in program debugging. It records the runtime state information for each program source line executed. It is coarser-grained and more efficient than the quadruple recording.
3. Procedure call/return recording. This is the coarsest-grained recording considered. It has the lowest time and memory space overhead among the three. However, execution information is not available inside procedure executions. For example, if a procedure consists of a loop in which there is an assignment to variable x, this recording mode would only save the last value of x and destroy all of the previous values in the loop.

Each of these basic recording granularity can be applied with an integer parameter *n*, which specifies that only one recording is to be made for each *n* steps. For example, runtime state can be recorded every other source line, using the source line recording and parameter 2. This gives us many possibilities for choosing the way that program execution history is recorded. Nevertheless, multiple recording granularities supported in our current design just define a starting point in improving the scalability of persistent runtime systems. More sophisticated options will be discussed in Chapter 8.

4.2.1.3 Transparency

Transparency becomes an issue at two levels. It is easiest to hide execution history recording from runtime system users. Then only the performance penalty of persistency will be visible to system users. In other regards, the persistent runtime system will look just like an ordinary ephemeral runtime system.

The transparency is also available at a lower level in the persistent runtime system. We separate the memory component of our system from its SETL interpreter, making persistency confined to the memory component and transparent to the language

interpreter. Thus, changing one part of the system will have minimal impact on the other.

4.2.1.4 Performance

Performance is always a bottom line issue. Too slow a persistent runtime system would make our debugging approach impractical, no matter how successful it might be in theoretical terms. As we have seen in early discussion, one of the major goals of our lazy debugging system is a persistent runtime system which does not incur excessive in execution time or memory space overhead. We are therefore concerned to compare the performance of our persistent runtime system with that of an ephemeral runtime system of similar design.

But more fundamentally, the persistent runtime system must be kept simple so that we can concentrate on its innovative aspects without becoming much involved in detail. We need speed of prototyping and ease in maintenance, so that we can get early feedback to verify the soundness of the design and make necessary modifications. Thus, many possible efficiency enhancements have been deferred in the expectation that an implementation with higher performance can be designed later.

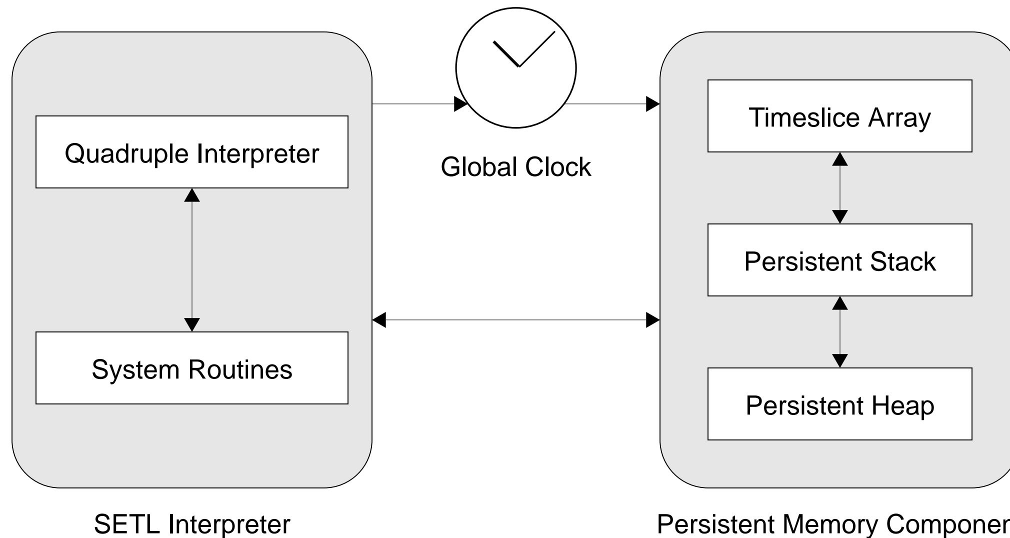
4.2.2 System Design

As Dijkstra remarks, "The technique of mastering complexity has been known since ancient times: divide et impera (divide and rule)"[22]. In the design of our SETL persistent runtime system, we have used an object-oriented decomposition[12]. We decompose the system into smaller and smaller parts, each of which is an autonomous object supporting some higher level behavior. Figure 4-1 shows the system structures, which includes a SETL interpreter, a persistent memory component and a global clock.

4.2.2.1 Quadruple Interpreter and System Routines

Our SETL interpreter is conventional[1][6], and consists of a SETL quadruple interpreter and a library of system routines. Because execution history recording is handled internally to the persistent memory component, the quadruple interpreter and system routines closely resemble those used in ephemeral runtime systems: they simply

FIGURE 4-1: Structures of the SETL persistent runtime system



interpret the quadruples of a SETL program until the execution stops, either normally or after encountering a runtime error.

4.2.2.2 Timeslice Array

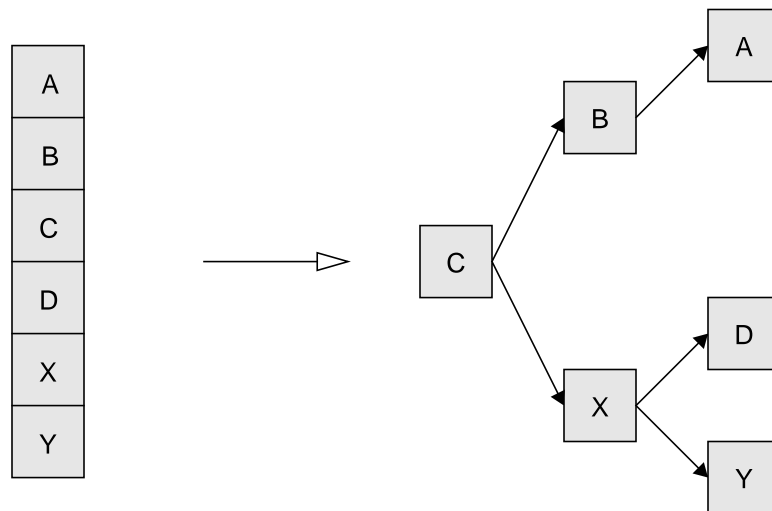
A timeslice array is used in the memory component, serving as the starting reference points to the valid execution information for each execution moment. The primary function of the timeslice array is to store execution stack pointers: the stack pointer at timeslice *i* is stored in the *i*th position of this array. This guarantees that any runtime memory operation can be initiated in $O(1)$ time. It is also used to store some time-related execution historical information, e.g. the line number of the source line being executed at a timeslice and the number of quadruples that have been interpreted so far.

4.2.2.3 Persistent Stack

A persistent stack is the first part of the persistent data structures that are used in our persistent memory component. Like an ordinary execution stack, it keeps track of procedure activations and provides storage or access for local variable objects. Conventional compilers or interpreters usually store local variable objects or their pointers in an array for efficient access*. Although a similar arrangement is possible in

our persistent runtime system, it would result in activation records of unbounded size, making persistent update very expensive. Therefore, we chose to break a conventional activation record into a number of smaller objects, as illustrated in Figure 4-2. We store local variable values in a persistent heap, and keep their pointers in an activation record. Although such an arrangement introduces logarithmic time overhead for variable access, it allows our system to perform persistent updates locally, and thus achieves a better memory usage performance. This design decision is made solely based on memory efficiency considerations and somehow arbitrary. We can switch back to the normal approach if time efficiency is of primary concern.

FIGURE 4-2: Breaking a large data object into several pieces



Local variables array in an activation record are broken into a tree of smaller chunks

The persistent stack also needs to keep track of changes in its structure. It records the result of each push and pop operation, as well as every update of variable value pointers propagated by variable updates.

*. SETL compilers and interpreters only store pointers because SETL object size can not be determined statically.

4.2.2.4 Persistent Heap

Persistent heaps are the second part of the persistent data structures used in our system. They are more complicated than the persistent stack, because they manage allocation, storage, and deallocation of the variable values used in a program execution. They must provide structures for storing and accessing variable values, and also record variable value history to fulfill its persistency requirement. An update occurring in a node in persistent heaps may cause cascading updates in nodes of upper levels or of the persistent stack.

4.2.2.5 Global Clock

We use a global clock to represent the advance of program execution and to signify new runtime states. It is a logical clock shared by the SETL interpreter and the persistent memory component. Its value is determined by program execution speed and the recording granularity selected, and is used by the memory component to access and update appropriate runtime values.

4.3 Implementation

The implementation of our SETL persistent runtime system starts with the clock and persistent memory component and is generally straightforward. It uses the *node splitting* method with minor extensions. Instead of insisting on a *purely persistent* system, we allow each node in the persistent execution stack and persistent heaps to perform both persistent updates and ephemeral updates, depending on the temporal relation between the timestamp of the node and the global clock value. A node performs an ephemeral update (overwrite) if its timestamp is equal to the global clock value, but a persistent update otherwise. This allows us to vary the granularity of our system recording.

4.3.1 Global Clock

We implement the global clock as a global counter with initial value 0. The value increases when execution enters a new timeslice. The recording granularity selected determines the clock speed, which can be as fast as increasing 1 for each quadruple interpretation, or as slow as remaining 0 for a whole execution.

4.3.2 Timeslice Array

We implement the timeslice array as a vector of some fixed length (1000 is the current initial length). Once the capacity limit of the vector is reached, our system allocates a new vector of double length, copies the values from the old vector, and uses it thereafter until its capacity limit is reached.

4.3.3 Persistent Stack

We implement the persistent stack using a persistent linear LIFO list, consisting of two classes of objects: namely stack nodes and frame nodes. A frame node stores the actual information of an activation record, while a stack node is a persistent node, containing a frame node and maintaining the structure of persistent linear list.

Figure 4-3 shows the data representation of stack nodes and major methods associated with them. A stack node is an 8-field vector in Scheme, obtained by augmenting an ordinary linear list node with some additional fields required by the *node splitting* method: a timestamp, an extra pointer, a timestamp for the extra pointer, and a copy pointer.

FIGURE 4-3: Data representation of stack nodes and their major methods

```
; data representation of stack node
; #(stack-node-typetag
;   timestamp
;   head-pointer
;   stack-frame
;   tail-pointer
;   extra-pointer
;   timestamp-for-extra-pointer
;   copy-pointer)

; major methods on stack node
; make-stack      allocate and return a new stack node
; top-stack       return the frame of top stack node
; pop-stack!      pop top stack node and return its frame
; push-stack!     push a stack node
; insert-stack!   insert a stack node
; delete-stack!   delete a stack node
; update-stack!   update a stack node when its frame changes
```

Push-stack!, pop-stack! and top-stack are the three key methods. These may invoke insert-stack! or delete-stack!. The implementation of insert-stack! uses a persistent update of the *node splitting* type, while delete-stack! is implemented by inserting an empty stack node into the parent of the node to be deleted. Update-stack! allocates a new stack node to accommodate the updated values, and inserts the newly allocated stack node to the original parent.

Figure 4-4 shows the data representation of frame nodes and major methods associated with them. A frame node is a vector of 11 fields, storing the actual information of an activation record, including program counter, procedure name, pointers to global, local, and temporary variables, pointer to calling parameters, pointer to return parameters, and an additional field for a set of special quadruple instructions.

FIGURE 4-4: Data representation of frame nodes and their major methods

```
; data representation of frame node
; #(stack-frame-typetag
;   parent-pointer
;   pc
;   procedure-name
;   global-variable-pointer
;   local-variable-pointer
;   temporary-variable-pointer
;   parameters-in
;   parameters-out
;   return-value
;   additional-field)

; major methods on frame node
; make-frame      allocate and return a new frame node
; access-frame    access a variable on a frame node
; insert-frame!   update a variable on a frame node
; update-frame!   update a frame node when its variable pointers change
```

Access-frame and insert-frame! are the two key methods for accessing and updating program variables respectively. To update a frame node, update-frame! allocates a new frame node to accommodate the updated values, and invokes update-stack! in the containing stack node of the frame node.

4.3.4 Persistent Heap

Persistent heaps can be implemented in several ways, e.g., as persistent linear lists, persistent binary trees, persistent balanced trees, persistent B-trees, and so on. We have selected the persistent binary search tree for its simplicity and efficiency. Like the persistent execution stack discussed in the preceding section, persistent heaps are implemented using two classes of objects: heap nodes and variable nodes. A variable node stores the actual information for a SETL variable, while a heap node is a persistent node, which maintains the structure of persistent binary search trees.

Figure 4-5 shows the data representation of heap nodes and major methods associated with them. The data representation of a heap node is a 10-field vector, obtained by augmenting an ordinary binary search tree node with the following additional fields: a timestamp, an extra pointer, a timestamp for the extra pointer, a state field of the extra pointer specifying whether the extra pointer is used as a left child or as a right child, and a copy pointer.

FIGURE 4-5: Data representation of heap nodes and their major methods

```
; data representation of heap node
; #(heap-node-typetag
;   timestamp
;   parent-pointer
;   variable-pointer
;   left-child-pointer
;   right-child-pointer
;   extra-pointer
;   timestamp-for-extra-pointer
;   state-for-extra-pointer
;   copy-pointer)

; major methods on heap node
; make-node      allocate and return a new heap node
; access-node      access a heap node in a tree
; insert-node!      insert a heap node into a tree
; delete-node!      delete a heap node from a tree
; update-node!      update a heap node
```

Insert-node! and delete-node! are the key methods for inserting a heap node and deleting a heap node respectively. As in the delete-stack! operation for stack nodes, delete-node! is

implemented by inserting an empty heap node into the parent of the node to be deleted, after those relevant links are properly updated as required when deleting an ordinary binary tree node.

Figure 4-6 shows the data representation of variable nodes and major methods associated with them. A variable node is a 5-field vector, storing the actual SETL variable information, which includes a pointer to its parent node, plus variable name, variable type, and variable value fields. Unlike stack nodes and frame nodes, heap nodes and variable nodes in persistent heaps can be mutually recursive, in the sense that a heap node always contains a variable node, and a variable node can contain a heap node when its value is of SETL compound type.

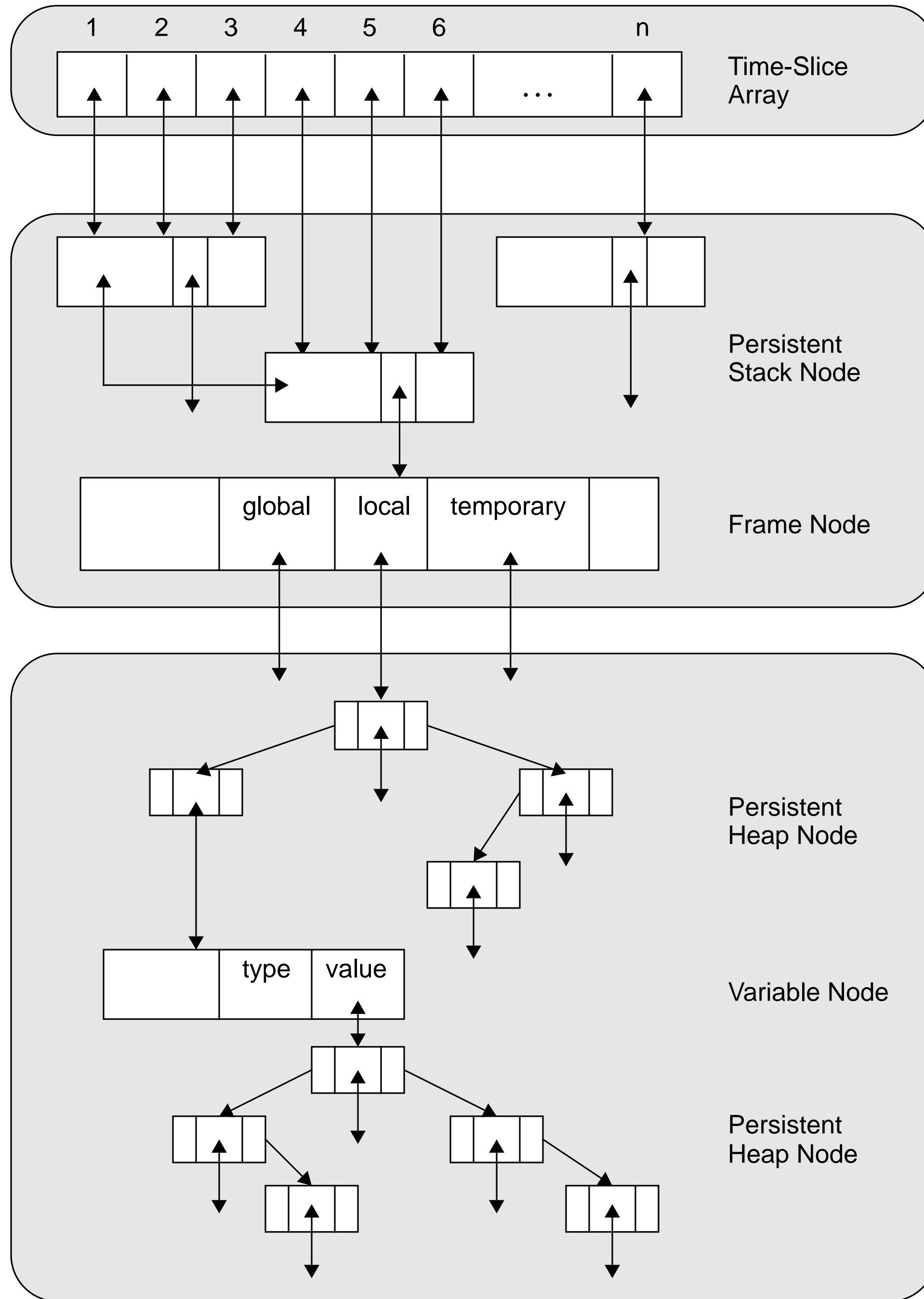
FIGURE 4-6: Data representation of variable nodes and their major methods

```
; data representation of variable
; #(variable-typetag
;   parent-pointer
;   variable-name
;   variable-type
;   variable-value)

; major methods on variable
; make-variable       allocate and return a new variable node
; access-variable     access a heap node in variable node
; insert-variable!    insert a heap node into variable node
; delete-variable!    delete a heap node from variable node
; update-variable!    update variable node when its type or value changes
```

Overall, the memory component of our SETL persistent runtime system can be viewed as a data structure constituting of three levels, as shown in Figure 4-7. The persistent nodes used in this component are organized in such a way that they satisfy an essential condition of the “node splitting” method: a node points only to nodes that cover a subrange of its time range.

FIGURE 4-7: Structures of the memory management component



4.3.5 Quadruple Interpreter and System Routines

Because execution history recording is designed to be transparent to the SETL interpreter, its implementation is standard, and closely follows a conventional compiler/interpreter approach. The interpreter is therefore almost identical to that for an ordinary runtime system. One interesting difference is so called *variable shifting* phenomenon in the persistent runtime system, which caused a very subtle bug during development. Variable shifting may occur when the interpreter obtains a variable pointer from the memory component, modifies the variable, and then erroreously retrieves the old variable value using the same pointer. Actually, the new value is stored correctly, and is pointed to by a pointer copy, and the current pointer has become obsolete after the update. We simply added an additional interpreter check to solve this problem.

CHAPTER 5 More on User Interface Design

An overall survey of our debugger's interface was given in Chapter 3. This chapter provides more details of interface design and implementation. We begin by describing general design principles which shaped the interface, and then go on present several of its key implementation techniques.

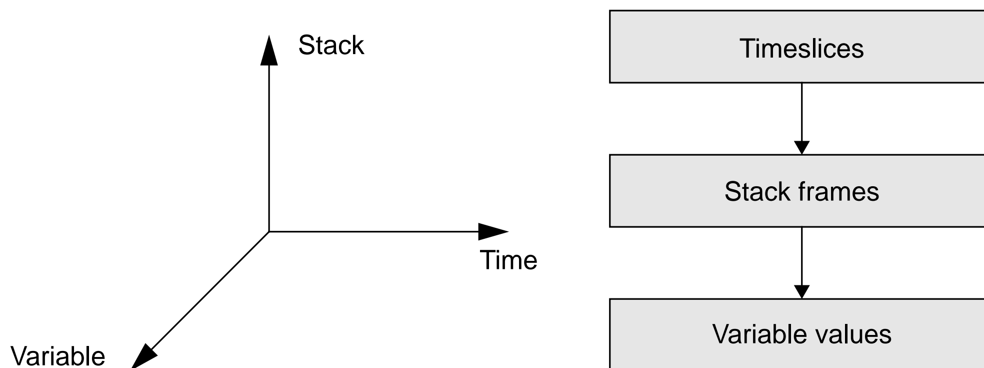
5.1 Overall Design Principles

As previously said, its graphical user interface is very important to LSD. The interface is designed to support the incremental debugging model discussed in Chapter 2. The aim of the interface is to allow users to explore it freely. Following suggestions derived from[59], we represent execution history using visual objects and allow users to debug programs by manipulating these objects. The interface appears to users as a coordinated collection of objects, each of them can be manipulated by mouse operations, e.g. "pointing", "clicking", and "dragging". These operations are rapid and reversible. As in other well-conceived interactive systems, our interface is designed to be simple to use, easy to comprehend, and consistent and predictable in behavior.

5.2 Execution Information Structures

As we have seen in Chapter 3, the interface of our debugger provides multiple views of execution history using multiple windows, in which each window shows a portion of execution history in a perspective. The organization of these windows reflects the conceptual organization of our debugging interface, as shown in Figure 5-1. We view execution history as a three-dimension information space, whose three dimensions are time, stack, and variable. At the top level, execution time divides execution history into timeslices. Each timeslice is divided into stack frames, which are then divided into variable values.

FIGURE 5-1: Organization of execution history



Our graphical user interface keeps overall execution information constantly available to users through the main window. This information provides a global view of program execution and serves as a starting point for any further, detailed data examinations. The interface allows users to examine any portion of this history by zooming into it. More specific execution information (e.g., execution stack and variable values at each timeslice) is stored in an underlying persistent runtime system. Most of this information is shown only when requested explicitly, by popping up the stack window and by applying interactive display facilities. Our interface focuses on the execution information relevant one timeslice. This allows users to concentrate on a small portion of whole execution history at any time.

5.3 Execution Trace Display

This section discusses the algorithm that used to display execution trace in the history window, which must display arbitrary number of trace marks in a limited window space. We give all trace marks the same height, which is the height of the font used in the source code window. The trace marks displayed at any given moment have a uniform width, which varies depending on the number of trace marks to be displayed and the width of the history window. The more trace marks to be displayed, the smaller their width. Depending on the number of the trace marks to be displayed, the width of a trace mark can be arbitrarily small, down to a single pixel. In practice, only a sample of the complete set of trace marks may be displayed. We distribute trace marks samples uniformly across whole execution history, and require that the width difference between any two trace marks displayed does not exceed one pixel, and that the trace marks with different widths are uniformly distributed in the window.

FIGURE 5-2: Efficient execution trace display algorithm

```
procedure DISPLAY_TRACE(W: in WINDOW; M, START, END: in INTEGER; T: in INTEGER_ARRAY) is
var
   N, SHORT, LONG, LOC, LOWER, UPPER, I, WID, DELTA: INTEGER;
begin
   N := END - START + 1;
   SHORT := FLOOR (M/N);    LONG := SHORT + 1;
   LOWER := N * SHORT;    UPPER := LOWER + N;
   LOC := 0;    DELTA := 0;
   for I := START to END loop
      if (M + DELTA) - LOWER <= UPPER - (M + DELTA) then
         WID := SHORT;    DELTA := DELTA + M - LOWER;
      else
         WID := LONG;    DELTA := DELTA + M - UPPER;
      end if;
      if LEN > 0 then
         DISPLAY_MARK (W, LOC, WID, T[I]);
         LOC := LOC + WID;
      end if;
   end loop;
end DISPLAYTRACE;
```

A straightforward but naive implementation of this algorithm would require $O(n)$ floating-point multiplications, $O(n)$ floating-point additions, and $O(n)$ floor operations, which is too expensive for quick update, as often required when a scrollbar is dragged in

the history window. Figure 5-2 shows pseudo Ada code for the more efficient display algorithm that we have developed, in which DISPLAY_MARK is a low-level graphical routine that displays a rectangle of width wid at position (loc, T[i]) in window W. This is an incremental algorithm, and only requires $O(1)$ fixed-point multiplications, $O(n)$ fixed-point additions, and $O(1)$ floor operations. For each trace mark to be displayed, this algorithm computes a location value loc and width value wid by maintaining appropriate delta, the difference between the location computed and its ideal value in a display space with infinite precision. The algorithm uses the following three recursive expressions:

$$loc_{i+1} = loc_i + wid_i \text{ and } loc_{start} = 0 \quad \text{(Eq 5.1)}$$

$$\delta_{i+1}/n = \delta_i/n + m/n - wid_i \text{ and } \delta_{start}/n = 0 \quad \text{(Eq 5.2)}$$

$$wid_i = \lfloor m/n + \delta_i/n + 0.5 \rfloor \quad \text{(Eq 5.3)}$$

where *start* and *end* are the starting and ending timeslices of the execution history to be displayed respectively, $n = end - start + 1$, is the number of trace marks to be displayed, m is the history window's width in number of pixels, and $start \leq i \leq end$.

5.4 Interactive Display

Interactive display, an interesting facility supported by our interface, is the most complex one to implement. This is largely because that interactive display must map users' gestures to subvalues of a compound SETL value. Our implementation was simplified by the fact that SETL values do not have problems of pointer aliasing, so that each sub-value has a unique access path.

We have implemented interactive display in the following four steps: 1) organizing SETL variable values as tree-like data structures; 2) mapping display layouts for these values geometrically into a text window; 3) locating the value that users manipulate by following these geographical mappings; 4) performing display on the value located.

First, we organize the accessible variables at an current timeslice using a linear list. The first part of the list consists of local variables and the second part of global variables.

Variables in each part of the list are sorted alphabetically by their names. Next, we organize the value of a variable as a tree of nodes having arbitrary fan-out. Each simple SETL value is a leaf node in a value tree while each compound SETL value is an internal node (see Figure 3-8). Each node includes the information necessary for its display, including its printing mode, tree level, and string representation. Figure 5-3 shows these data representations in pseudo Ada code.

FIGURE 5-3: Data representations of variable values

```
type SETL_TYPE is (SETL_ATOM, SETL_BOOLEAN, SETL_INTEGER, SETL_OM,
                   SETL_REAL, SETL_SET, SETL_STRING, SETL_TUPLE);
type PRINT_MODE is (LSD_PRINT, LSD_DISPLAY, LSD_ZOOM_IN, LSD_ZOOM_OUT);

type VARIABLE;
type VARIABLE_POINTER is access VARIABLE;
type VARIABLE_ARRAY is array (NATURAL range <>) of VARIABLE_POINTER;
type VARIABLE is
  record
    NAME: STRING;
    TYPE: SETL_TYPE;
    NUMBER_OF_CHILDREN: NATURAL;
    CHILDREN: VARIABLE_ARRAY;
    LENGTH_OF_VALUE: POSITIVE;
    VALUE: STRING;
    MODE: PRINT_MODE;
    LEVEL: NATURAL;
  end record;

type VARIABLE_LIST;
type VARIABLE_LIST_POINTER is access VARIABLE_LIST;
type VARIABLE_LIST is
  record
    VAR: VARIABLE;
    NEXT: VARIABLE_LIST_POINTER;
  end record;
```

Geometric mappings of variable values into a text window obey the following rules: Variable names are listed alphabetically from window top to window bottom, local variables first and global variables second, and each name occupies one line. A variable value, when printed, is inserted into the lines below its name. A variable value in Print mode is shown in one line, as for the SETL print statement. In Display mode, a variable value is shown as a value tree in which each simple data item occupies one line and has proper indent, if the value is of compound types, or is the same as that in Print mode

otherwise. A variable value in Zoom In mode is listed as "[...]" or "{...}" in one line if it is a non-empty compound value, or is the same as that in Print mode otherwise.

These rules potentially allow a variable to be printed in a large number of ways. To manage its display during user manipulations, the interface maintains two invariant properties. 1) The place in which a value is printed is determined only by the printing modes of prior variables and the printing modes of the values of the upper levels in the same variable. 2) The way that a value is printed is determined by the printing attributes of the value itself (i.e., its printing mode, its level in the variable tree, and its string representation). These properties allow the interface to interpret user manipulations uniquely and correctly.

The implementation uses two principal routines: Locate_Value(Position, Data_Structure) and Print_Value(Variable, Printing_Mode). Locate_Value takes two parameters: a mouse clicking position in the stack window and a data structure of accessible variables. It traverses the variable data structure, and returns the value manipulated by users. PrintValue is then called with the value located and the printing mode associated. It preforms the appropriate printing operation.

CHAPTER 6 Internal Structure of the System

This chapter describes the internal routines of our Lazy SETL Debugger, which serve as a layer connecting the debugger's graphical user interface and its persistent runtime system. We describe key design considerations, communication protocols supported, and some additional implementation issues.

6.1 Design Considerations

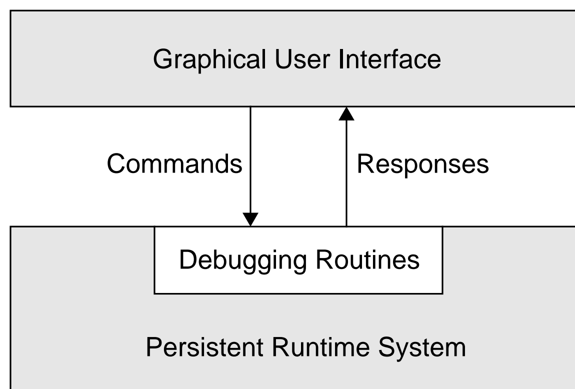
The PRS and GUI of our debugger are logically independent processes, designed as two separate programs. They are only connected in LSD by a communication protocol, defined by a number of special routines. We refer to these routines as *debugging routines* because their primary function is to make the execution history recorded by the runtime system available to the graphical interface, and to support a variety of debugging operations, e.g., graphical program animation and execution history refinement.

If we chose to view its persistent runtime system as a debugging target and its graphical user interface as the whole debugger, LSD would come to have an organizational structure similar to that of some conventional breakpoint debuggers that operate in the UNIX environment[9]. These debuggers usually fork their debugging targets as child processes, which run in debugging mode. The debugger process and debugging target

process have different name spaces in such an organization, and one must allow the debugger process to control the execution of debugging target process, which must make its execution information accessible to the debugger. This is usually accomplished by using the UNIX ptrace routines[9]. Our LSD approach is much simpler, mainly because of the interpretive implementation of the persistent runtime system used. Instead of using general but low-level ptrace routines, we have designed a custom collection of debugging routines that achieve similar functionality.

Ideally, the debugging routines should be designed as an independent layer between the runtime system and its interface. In practice, it is more convenient and considerably more efficient to design these routines as a stub within the runtime system, as shown in Figure 6-1. Such an arrangement allows the debugging routines to share the same name space as the persistent runtime system so that we can avoid a costly context switch during communication. This arrangement does not introduce any serious compromise in system modularity or information encapsulation, as the debugging routines stub functions exactly the same way as a separate layer to the debugger's user interface.

FIGURE 6-1: Debugging routines layer as a stub in persistent runtime system



6.2 Communication Protocol

We have designed a protocol to define the communication supported by the debugging routines. This is a two-way protocol consisting of two parts: a command protocol and a

response protocol. Functionality, reliability, and simplicity are our primary concerns in design.

FIGURE 6-2: Command protocol

```
           command  →  load file_name
                    |  run
                    |  step
                    |  continue
                    |  stop
                    |  granularity granularity_type granularity_parameter
                    |  collapse time_range
                    |  stack timeslice frame_level
                    |  locate variable_name direction timeslice time_range
                    |  animation animation_type expression timeslice

         file_name  →  string
  granularity_type  →  quadruple | source line | procedure
granularity_parameter → number
         timeslice  →  number
       frame_level  →  number
     variable_name  →  string
         direction  →  forward | backward
        time_range  →  timeslice timeslice
    animation_type  →  points | lines | strings
        expression  →  string
```

Typically, the debugging routines stub receives a command (from the interface) and responds. Figure 6-2 and Figure 6-3 give a context-free grammar for the command protocol and corresponding response protocol respectively. The command protocol currently supports 10 commands. The load command takes a file name parameter, translates the program into quadruples, and loads these into the persistent runtime system. The granularity command sets our debugger's recording granularity. The run, step and continue commands put the debugging target into execution, and return an execution trace covering the run up to its last execution moment. The stop command stops the debugging target's execution and also returns an execution trace. Detailed

FIGURE 6-3: Response protocol

load	::	OK
granularity	::	OK
run	::	*trace_size trace** OK
step	::	*trace_size trace** OK
continue	::	*trace_size trace** OK
stop	::	*trace_size trace** OK
collapse	::	*trace_size trace** OK
stack	::	*stack_depth procedure_name* local *var** global *var** OK
locate	::	*trace* OK
animation	::	*animation_value** OK
trace_size	→	**number**
trace	→	**number**
stack_depth	→	**number**
procedure_name	→	**string**
var	→	*var_name type_value*
var_name	→	**string**
type_value	→	*simple_type value* | *compound_type cardinality type_value_list*
simple_type	→	atom | boolean | integer | om | real | string
value	→	**string**
compound_type	→	set | tuple
cardinality	→	**number**
type_value_list	→	ε | *type_value type_value_list*
animation_value	→	**string**

execution information can be accessed by using the stack, locate or animation commands. Stack takes a current timeslice and a stack frame level, and returns complete information for the stack frame specified at the timeslice. This information includes stack depth, activating procedure name, stack frame level, local and global variable values. Locate takes a variable name, a search direction, a current timeslice and a timeslice range. Depending on the search direction, it returns the closest timeslice in the timeslice range in which the variable's value changes. Although locate is not essential to the communication protocol in the sense that we can simulate it by a sequence of stack commands, it is nevertheless included to reduce communication overhead. Animation takes an animation type, a SETL expression and a current timeslice, and registers a graphical program view. Once registered, this view will be activated upon each of following stack commands. The value returned for the view is the value of the view expression in ASCII format. Collapse takes a portion of execution history, re-executes it using the current recording granularity, and returns a new execution trace after re-execution is complete.

6.3 Implementation Issues

6.3.1 Implementation Overview

The persistent runtime system was implemented on a Sun SPARCstation using Scm, a public-domain interpretive Scheme implementation with a conventional *mark-sweep* garbage collector[81]. The runtime system supports a full SETL implementation including backtracking, with a few simplifications and modifications in I/O operations. The implementation consists of approximately 7000 lines of Scheme code.

The starting point for the implementation was the SETL translator written by David Bacon. This translates a SETL program into a sequence of quadruples for interpreted or compiled execution. Translations are basically done line by line and use few local or global optimizations. Although the quadruples generated do not attain high execution speed, they are perfect for debugging purposes because the quadruples preserve full symbolic information from the original SETL program. Another useful feature of the translator used is that it does not require its input to be a complete program, but it can translate even a single valid SETL source line. This allows us to evaluate (animate) arbitrary SETL expressions during debugging. This translator is used as the front end of our persistent runtime system, which reads, parses, links, and interprets the quadruples generated.

The graphical user interface of our debugger was implemented using C and the OPEN LOOK widget set (also known as OLIT, Open Look Intrinsic Toolkits) developed by AT&T and Sun Microsystems[76]. Like other professional level user interface toolkits such as Motif[25], OPEN LOOK provides a set of graphical widgets with which programmers can implement attractive and consistent graphical user interfaces without worrying much about X window details. The OPEN LOOK widgets used in our system include text windows, draw windows, popup windows, pulldown menus, popup menus, scrollbars, scrolling lists, buttons, text fields, cursors, as well as many invisible manager widgets that control the layout of visible objects. The implementation requires approximately 7000 lines of C code.

6.3.2 Communication Support

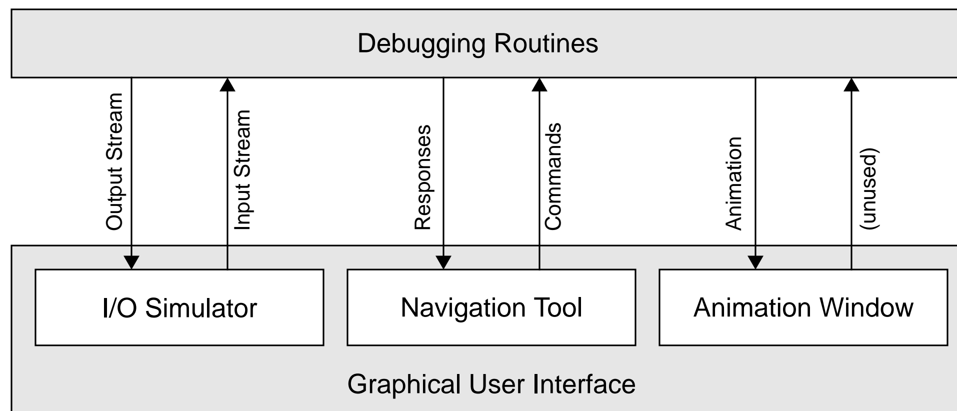
The communication protocol supported by the debugging routines was designed to be based on standard communication channels, e.g., UNIX pipes, message queues, semaphores, shared memory, sockets, etc. Deferring many engineering considerations, we implemented our communication protocol using pipes. More specifically, we implemented the graphical user interface of LSD as a child process of the debugging routines stub, and connected to it using three pairs of pipes, as shown in Figure 6-4. Commands and responses are transmitted over the pipes as byte streams, which are parsed at both ends using predictive parsers. We implemented the pipe connections by adding two C routines into the Scm environment, which can be used as Scheme's essential procedures[2]:

(open-rw-pipes *string*) procedure

(close-rw-pipes *pipe-pair*) procedure

Open-rw-pipes takes a string argument that specifies an object code, creates a pair of pipes, forks the object code as a child process, redirects its standard input and standard output to the pipes created, and returns them as a *pipe-pair* on success. We can close such a *pipe-pair* by using close-rw-pipes.

FIGURE 6-4: Communication protocol implemented using three pairs of pipes



6.3.3 Program Animation

As we have seen from the preceding chapter, our debugger can graphically animate SETL expressions during program debugging. Program animation is supported primarily by the animation and stack debugging routines. For an expression *E* to be animated, animation creates a one-line SETL program "print (*E*)", translates this program into quadruples, registers the quadruples for animation, and pops up a corresponding animation window. Whenever the current timeslice changes, the debugger's interface activates the stack routine to fetch stack and variable information at the new timeslice. This routine also checks before return to determine if there is an animation view registered. If so, stack saves the context of debugging target's execution and evaluates the animation view quadruples in a new execution context (***animation view context***). A program execution context includes quadruples, symbol table, execution recording granularity, program counter, runtime memory, and so on. Animation view evaluation uses single source line recording granularity to assure that the values used in the evaluation come from the desired timeslice. During animation evaluation, any symbol used in animation view quadruples is first checked against debugging target's symbol table. If there is a match, the value is fetched from the execution context; if not, the value is fetched from the animation view context. Finally, the value produced by evaluation is sent to the animation view context's output, which connects to the standard input of the animation window.

6.3.4 Program Re-execution

As we saw from the debugging example in Chapter 3, execution history collapsing supports re-execution of program, starting from an arbitrary timeslice. Re-starting execution from a recorded runtime state is not hard. As in backtracking, we simply restore the runtime state at the desired timeslice as a starting runtime state and continue quadruple interpretation. Stopping re-execution at a specified timeslice is not quite so simple. Simply checking by comparing runtime states to determine whether re-execution has reached its stopping moment is complex and time-consuming. Moreover, this method may not uniquely identify this moment, because two different timeslices

may have identical runtime states. The technique we have developed for this is to enhance the persistent runtime system with a quadruple counter, which records the total number of quadruples interpreted from the beginning of execution. Given this information, re-execution only needs to interpret a number of quadruples equal to the difference between the number of quadruples that have been interpreted at the ending timeslice and the number at the starting timeslice.

CHAPTER 7 Performance

This chapter describes the performance of the system designed and implemented. We analyze the time and space performance, and the scalability of the persistent runtime system, and evaluate the debugging system using various usability issues.

7.1 Performance Analysis

This section analyzes the performance of our SETL persistent runtime system, focusing on three primary criteria: execution time, memory space usage, and scalability. We are interested in comparative performance of our system against a similar ephemeral runtime system, rather than in absolute figures. The ephemeral runtime system used in comparison was implemented using ephemeral linear FIFO list and ephemeral binary search trees. Actually this system was implemented first, from which our persistent runtime system was then obtained by switching to the corresponding persistent data structures.

We tested both the systems using a test set consisting of three SETL programs with different time and space complexities (see Appendix B). The first program is a text scanner that reads and prints SETL quadruples from a text file. This has a time cost of $\Theta(n)$ and a space cost of $\Theta(1)$, in which n is the number of quadruple lines in the text

input file. The second program computes prime numbers up to n, and has a time cost of $O(n\log n)$ and a space cost of $\Theta(n)$. Our last test program is an all-solution n-queens program, having an exponential time cost and a $\Theta(n)$ space cost. Performance figures are measured on a SUN SPARC 10 with 64 megabytes of main memory.

7.1.1 Time Performance

The figures measured for time performance include total execution time, garbage collection (GC) time, and the numbers of CONS operation performed by Scheme. These figures are provided by Scm, the implementation environment of our runtime systems.

Table 7-1 shows the total execution time figures in milliseconds of both the persistent runtime system and the ephemeral runtime system. In spite of relatively low execution speed, the results are encouraging when we compare the two runtime systems. Apparently the ratios of the total execution time between the persistent runtime system and the ephemeral runtime system are not high, ranging from 1.78 to 4.33. We also see that the ratios grow moderately when program input size increases.

TABLE 7-1: Total execution time of test programs (in milliseconds)

Scanner	**249**	**453**	**967**	**2672**	**4315**
ephemeral	5750	10316	21100	57966	91133
persistent	12850	28916	69800	190766	394900
pers / ephe	2.23	2.80	3.31	3.29	4.33
Prime Numbers	**100**	**200**	**300**	**400**	**500**
ephemeral	2183	5900	10750	17800	25200
persistent	4533	14766	32033	48666	74233
pers / ephe	2.08	2.50	2.98	2.73	2.95
N-Queens	**4**	**5**	**6**	**7**	**8**
ephemeral	4183	13150	44450	172200	729416
persistent	7450	31900	121783	605150	2892466
pers / ephe	1.78	2.43	2.74	3.51	3.97

The next important measurement for time performance is the GC execution time, which is shown in Table 7-2. Garbage collection in Scheme can consume a substantial portion of the total execution time. Compared to the total execution ratios, the GC time ratios between the two runtime systems are more significant, ranging from 2.17 to 10.07, and show a more rapid growth as input size increases.

TABLE 7-2: GC time of test programs (in milliseconds)

Scanner	**249**	**453**	**967**	**2672**	**4315**
ephemeral	1633	3033	6450	16283	25000
persistent	5316	14300	39533	106533	251666
pers / ephe	3.26	4.71	6.13	6.54	10.07
Prime Numbers	**100**	**200**	**300**	**400**	**500**
ephemeral	650	1983	3583	6333	9033
persistent	1783	7016	18900	25283	42900
pers / ephe	2.74	3.54	5.27	3.99	4.75
N-Queens	**4**	**5**	**6**	**7**	**8**
ephemeral	1350	4316	15516	62400	295800
persistent	2933	16633	71783	411350	2065266
pers / ephe	2.17	3.85	4.63	6.59	6.98

Mark and sweep, the GC method used by Scm, when invoked, suspends program execution temporarily, traverses and marks every memory block in use, and scans the entire memory to deallocate any unused ones. This approach is not effective in the case of the persistent runtime system implemented, and can consume approximately 30% - 70% of the total execution time on GC. Because persistent runtime systems use a lot of memory space to record execution history, this method often traverses a lot memory but can collect only very little garbage. Taking into account of the execution pattern of persistent runtime systems, we believe that using a custom GC routine or a *reference count* method would significantly reduce the GC time, and thus the total execution time as well.

Since it includes a very significant but largely avoidable portion of GC execution, the total execution time clearly does not accurately reflect the time performance attainable

by our approach. A more accurate measurement in this regard is actual execution time, defined as the difference between total execution time and GC time. Table 7-3 shows the actual execution time figures for our test programs. We can see that these ratios are low and stable, ranging from 1.59 to 2.17.

TABLE 7-3: Actual execution time of test programs (in milliseconds)

Scanner	**249**	**453**	**967**	**2672**	**4315**
ephemeral	4117	7283	14650	41683	66133
persistent	7534	14616	30267	84233	143234
pers / ephe	1.83	2.01	2.07	2.02	2.17
Prime Numbers	**100**	**200**	**300**	**400**	**500**
ephemeral	1533	3917	7167	11467	16167
persistent	2750	7750	13133	23383	31333
pers / ephe	1.79	1.98	1.83	2.04	1.94
N-Queens	**249**	**453**	**967**	**2672**	**4315**
ephemeral	2833	8834	28934	109800	433616
persistent	4517	15267	50000	193800	827200
pers / ephe	1.59	1.73	1.73	1.77	1.91

The accuracy of this production of actual execution time is further supported by the numbers of CONS operations performed by the Scheme interpreter, which are shown in Table 7-4. CONS is the most important and fundamental operation in Scheme, upon which many other essential functions easily can be simulated or are actually implemented. Numbers of CONS operations performed are good approximations of actual execution time. We can see that the ratios of the number of CONS operations performed, ranging from 1.63 to 2.09, match closely to those of the actual execution time computed.

In sum, Figure 7-1 depicts the four time performance ratios that we have discussed so far. It is easy to see that our SETL persistent runtime system is approximately half the speed of the ephemeral runtime system, which supports the time complexity results we obtained earlier in Chapter 4.

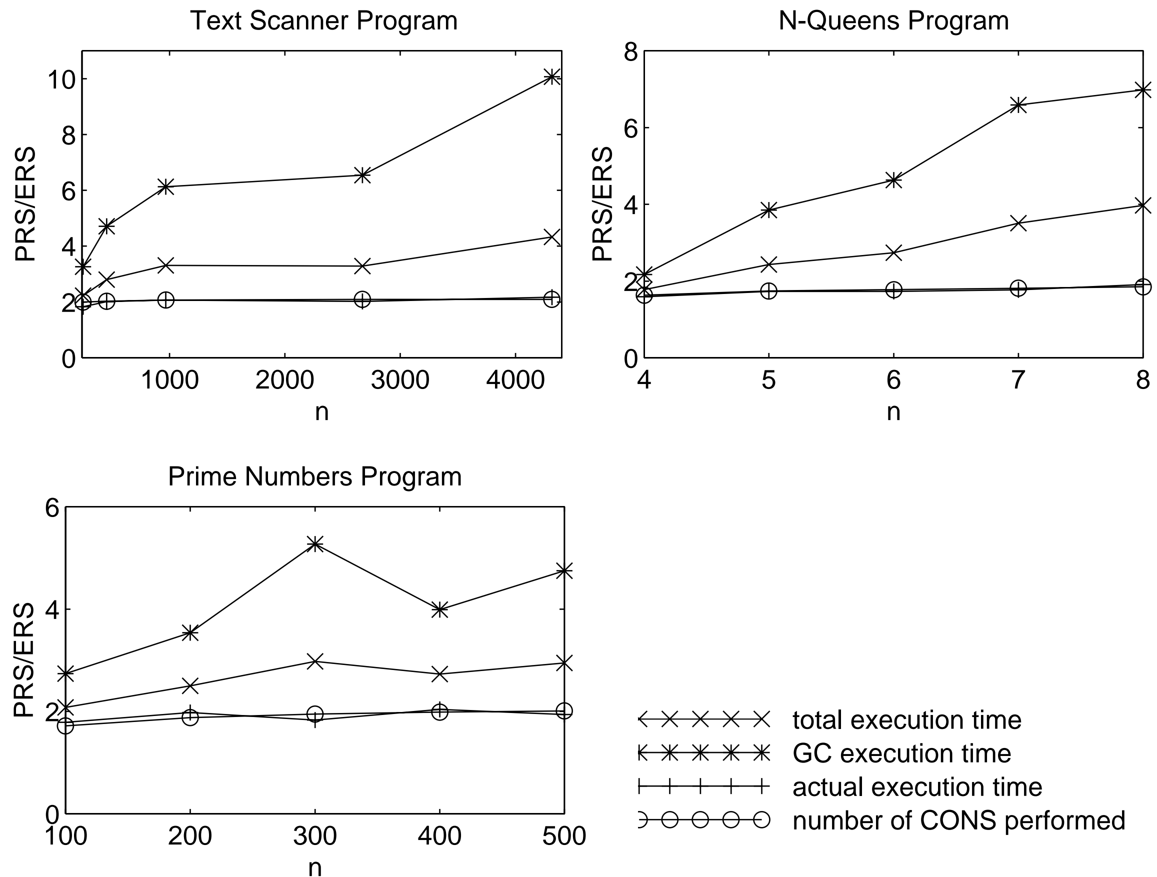
TABLE 7-4: Number of CONS operations performed of test programs

Scanner	249	453	967	2672	4315
ephemeral	1182207	2194171	4317815	11568840	19103793
persistent	2359654	4445765	8930010	24161731	39850250
pers / ephe	2.00	2.03	2.07	2.09	2.09
Prime Numbers	**100**	**200**	**300**	**400**	**500**
ephemeral	476202	1136658	2048471	3260787	4796062
persistent	817175	2141119	3998075	6492086	9658017
pers / ephe	1.72	1.88	1.95	1.99	2.01
N-Queens	**4**	**5**	**6**	**7**	**8**
ephemeral	966182	2858348	9307363	34447017	137047950
persistent	1578271	4984050	16586913	62415686	252876407
pers / ephe	1.63	1.74	1.78	1.81	1.85

7.1.2 Space Usage

The space performance figures are measured by the virtual memory consumed by the runtime systems, which includes the memory used for SETL source programs, quadruple codes, and runtime data structures. They are measured using a UNIX facility vmstat. Table 7-5 shows the figures and Figure 5-9 plots the ratios of these figures between the persistent runtime system and the ephemeral runtime system. Due to limitations of the measurement method used, these figures are not very precise (4-Kbyte is the smallest memory chunk that vmstat can measure). Nevertheless, they do give us a rough approximation of the memory usage. Figure 7-2 shows that there are moderate to substantial increases in the memory usage ratios. The exact figures vary from program to program, but apparently they do not have a sharper increase than $O(t)$, where t is a program's execution time. According to the complexity analysis we obtained earlier, the space costs of the three test programs would be $\Theta(t)$ for the text scanner program, $O(\log t)$ for the prime numbers program and $\Theta(t/\log t)$ for the n-queens program. These productions closely match the figures measured.

FIGURE 7-1: Time performance for test programs



7.1.3 Scalability

We are aware of the fact that the space overhead of our persistent runtime system can be substantial (e.g., the memory consumed by the text scanner program in the persistent runtime system is 2484 times larger than that used in the ephemeral runtime system). Therefore, we also test our system's scalability using different recording granularities. The text scanner program is used to study this issue because of the large memory requirement of its persistent version. The recording granularities used in testing are source line granularities with different parameters. Table 7-6, in which *n* is the granularity parameter, shows the actual execution time, the number of CONS operations

TABLE 7-5: Virtual memory size for test programs (in kilobytes)

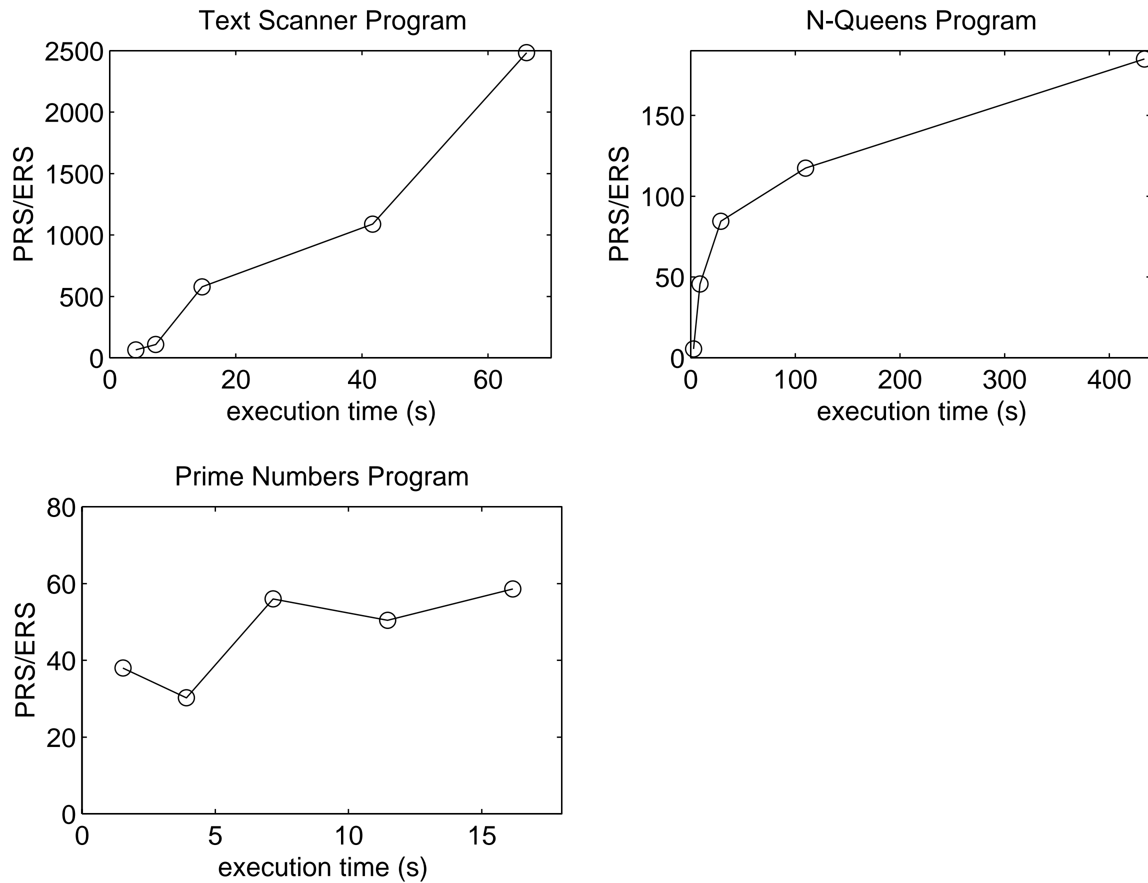
Scanner	4117 ms	7283 ms	14650 ms	41683 ms	66133 ms
ephemeral	4	4	4	4	4
persistent	260	536	2312	4356	9936
pers / ephe	65.0	109.0	578.0	1089.0	2484.0
Prime Numbers	**1533 ms**	**3917 ms**	**7167 ms**	**11467 ms**	**16167 ms**
ephemeral	4	20	44	68	80
persistent	152	604	2464	3428	4684
pers / ephe	38.0	30.2	56.0	50.4	58.6
N-Queens	**2833 ms**	**8834 ms**	**28934 ms**	**109800 ms**	**433616 ms**
ephemeral	8	12	36	100	236
persistent	44	548	3044	11744	43632
pers / ephe	5.5	45.7	84.6	117.4	184.9

performed and the memory space figures measured. For example, $n = 4$ represents the result recording one runtime state for the execution of 4 SETL source lines, and $n = 65536$ represents the coarsest-grained recording granularity, because only 23077 source lines are executed in the whole program.

TABLE 7-6: Time and memory consumed using different recording granularities

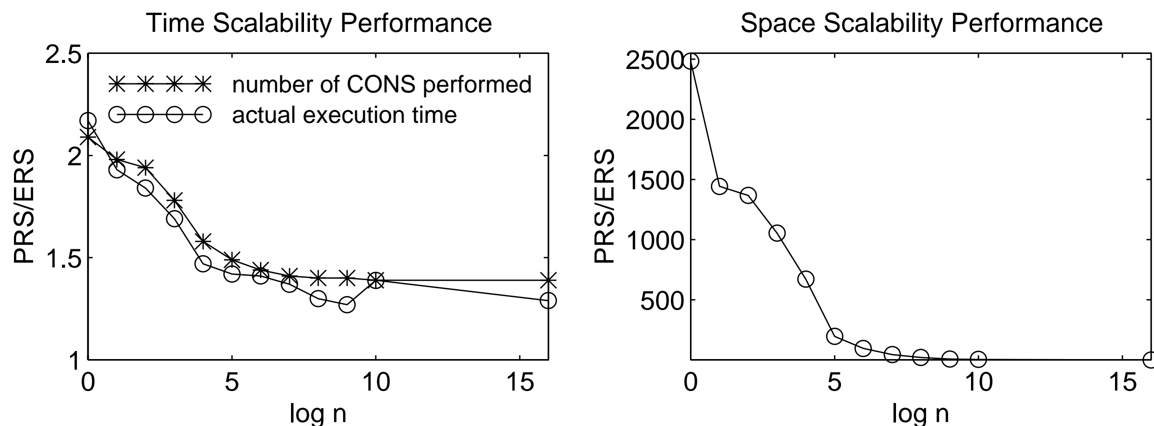
n	time (ms)	PRS/ERS	CONS	PRS/ERS	memory (kb)	PRS/ERS
1	143234	2.17	39850250	2.09	9936	2484.0
2	127330	1.93	37903239	1.98	5776	1444.0
4	121764	1.84	37155519	1.94	5468	1367.0
8	111647	1.69	33912917	1.78	4220	1055.0
16	97233	1.47	30270232	1.58	2692	673.0
32	93996	1.42	28405127	1.49	788	197.0
64	93058	1.41	27471597	1.44	384	96.0
128	90354	1.37	27004978	1.41	184	46.0
256	86084	1.30	26771821	1.40	84	21.0
512	83943	1.27	26652495	1.40	36	9.0
1024	91620	1.39	26593588	1.39	16	4.0
65536(∞)	85491	1.29	26459552	1.39	4	1.0

FIGURE 7-2: Memory performance for test programs



These figures show that our persistent runtime system can improve its both time and memory space performance gracefully by using coarse-grained recording granularities. For example, when a source line granularity of 64 is used, time overhead decreases from 2.17 to 1.41, and memory space overhead decreases from 2484 to 96. Nevertheless, in this case, we have still recorded 360 complete runtime states, and any possible re-execution is about 1/360 of a complete one. Figure 7-3 depicts these figures on a log scale of *n*. We can see that the memory space overhead diminishes rapidly as recording granularity parameter increases.

FIGURE 7-3: Scalability in time and space



7.2 Usability Evaluation

We now evaluate the usability of the debugging user interface implemented, employing the five usability attributes stated in [58]: learnability, memorizability, error rate, efficiency, and user satisfaction. Due to the prototype nature of the current implementation, thorough and complete evaluation of our system is hardly possible. The following discussion largely reflects comments gathered during several demonstrations rather than from a real testing environment.

1. Learnability. This graphical user interface is easy to learn. New users can usually start using it after 5 to 8 minute of explanation, although we do expect longer learning period when only a manual is available. In any case, all users have to learn are the operations for controlling objects in the interface, which includes five windows, approximately half dozen scrollbars, and several menus. In contrast, gdb has 160 commands falling into 10 categories, and dbx 52 commands falling into five.
2. Memorizability. The graphic nature and simplicity of our interface implies that it should be easy to remember. In fact, very little needs to be memorized, because the interface is simple, intuitive and self explanatory. Even its most innovative features

(i.e., its execution trace display and incremental printing) are straightforward and obvious, after initial explanation.

3. Error rate. Errors in using the interface often result from obvious mistakes such as trying to run a unloaded program. These errors are generally avoided by users with some experience of the system; novice users are provided with error messages. Once execution history has been recorded and our debugger enters the debugging stage, few errors are possible. Although a specific operation in data browsing or examination may not reveal any useful clue to bug location, it is harmless and trivially reversible in the interface.
4. Efficiency. The overall performance of the interface is very encouraging. The interface provides many powerful debugging functions not available or difficult to implement in conventional debuggers. These functions include backward stepping, backward control breakpoints, backward data breakpoints, and flexible data examination at arbitrary execution moments. Programs containing the common runtime errors (e.g., illegal operands) can usually be debugged very quickly. Even for programs having subtle bugs, our debugging interface helps users make progress naturally and easily. Our experience shows that a large portion of bugs can be found in constant time while others can be found in a length of time proportional to the program execution time. The efficiency of the operations supported in the interface is also good. Users normally receive feedback within a half second after issuing an operation. The primary bottleneck factors for higher debugging efficiency are the performance of the underlying persistent runtime system, communication delays, and the speed of typical user operations (e.g., moving mouse or clicking).
5. Satisfaction. Satisfaction ratings are particularly subject to users' non-objective judgement. Nevertheless, most people who have used or seen the debugger think that it is "very useful", "appealing", "impressive", etc.

7.3 Summary

The results we have described of our debugger are quite encouraging, taking into account of its prototype nature. It imposes only a modest slowdown in execution time and its memory requirements are manageable. A usability evaluation of the debugger interface gives promising conclusions since the system provides a number of easy to use, high-level facilities. All this indicates the practicality of our debugging approach.

CHAPTER 8 Open Issues

In this last chapter, we review several issues in the light of the results obtained in the preceding chapters. We begin by describing some limitations of our paradigm and the current implementation, and go on to suggest some extensions. We summarize our research by comparing our debugging system with conventional debugging approaches, and list some open issues for future work.

8.1 Current Limitations and Possible Solutions

Although the conclusions arrived at in the last chapter are encouraging, our approach has some significant limitations. Some of these limitations will become especially important when one attempts to build "industrial strength" lazy debuggers.

8.1.1 Theoretical Limitations

Although many of the limitations that we are going to discuss result from our design and implementation simplifications, and can usually be solved by putting in more engineering effort, some of them stem more fundamentally from our lazy debugging paradigm (i.e., the techniques used of making data objects persistent). Solutions of these

limitations will therefore depend on theoretical breakthroughs. These limitations include:

1. The *node splitting* method cannot efficiently implement all data structure operations. One example is the cardinality operation, which returns the number of nodes in a tree. Using ephemeral data structures, one can easily implement the cardinality operation in time complexity $O(1)$, by augmenting each node of the original data structures with a cardinality field and slightly modifying the update operation. However when *node splitting* persistent data structures are used, the time complexity of the cardinality operation is $O(n)$ instead of $O(1)$, where n is the number of nodes in the tree. The reason is that a persistent node can link to an arbitrary number of versions of data structures, so that the cardinality operation thus has to traverse the complete tree for a given timeslice. As it is an important operation in a SETL runtime (e.g., to access an indexed entry in a tuple or to compare two set values), this overhead is quite undesirable.
2. One of the key assumptions of the *node splitting* method is that each node in the data structures to be made persistent has only a fixed number of access points. This restriction does not affect SETL runtime systems because of SETL's value semantics. However, this nice property fails for a number of other programming languages that support pointer semantics, such as Ada, Pascal, and Scheme, for which the *node splitting* method is therefore not directly applicable. This major problem needs to be solved in trying to extend the lazy debugging paradigm to these languages.
3. All the above limitations could have been easily eliminated if arrays had efficient persistent representations (i.e., persistent access and persistent update operations in $O(1)$ time and space). The best results we know concerning persistent arrays is that reported by Dietz[21]. Dietz's method uses a context tree representation and has $O(\log \log m)$ overhead in both time and space, where m is the number of update operations performed on an array. It seems to us however that this method is very complicated and has a large constant factor, so it might not be a practical way of building efficient persistent runtime systems.

Having pointed out these limitations, we would like to revisit the four most important methods of making data objects persistent, and compare them using several important criteria as means for designing general-purpose and efficient persistent runtime systems. Table 8-1 summarizes an analysis of this issue. In the table, n is the size of the data objects to be made persistent; m is the number of update operations performed on the data objects; *last* is the newest version of the data objects; and *prev* is any of the previous versions. The most important point in this summary is the distinction made between the last version and previous versions of data object, because program execution takes place only in its *execution wave front* consisting of the last version of data objects. We notice that the *node splitting* method is the best one in terms of time and space overhead, but imposes strict requirements, such as the bounded number of access points and the data structures organization. However, the *fat node* method is more flexible and has the same performance results except for its $O(\log m)$ overhead in accessing previous versions of data objects. Thus suggests that a combination of the *node splitting* method and the *fat node* method may be superior to our current implementation.

TABLE 8-1: Comparisons among four methods of making data objects persistent

	Check-pointing	**Fat Node**	**Node Splitting**	**Persistent Array**
Space requirement	$O(mn)$	$O(m+n)$	$O(m+n)$	$O(m+n)$
Time requirement (access *last*)	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Time requirement (access *prev*)	$O(\log m)$	$O(\log m)$	$O(1)$	$O(\log \log m)$
Time requirement (update *last*)	$O(1)$	$O(1)$	$O(1)$	$O(\log \log m)$
Number of access points	unbounded	unbounded	bounded	unbounded
Organized as data structures	not required	not required	required	not required

8.1.2 Limitations of the Current Implementation

Ineffective garbage collection is the first and perhaps the most severe weakness of the current implementation of our Lazy SETL Debugger. Some mechanism for automatic reclamation of computer storage is always required in a SETL runtime system. For simplicity, the persistent runtime system of our debugger uses Scheme's runtime heap

as its heap, so that SETL runtime objects can always be garbage-collected by Scheme's garbage collector. This arrangement eliminates any need for a separate SETL garbage collector. The price we pay is that Scheme's garbage collector is very inefficient for our purposes (see Chapter 7). A major assumption of most current garbage collection methods is that "most objects live a very short time, while a small percentage of them lives much longer"[81]. Unfortunately, this assumption is completely wrong in the case of a persistent runtime system, which uses a major portion of its memory to hold execution history. Potentially better candidates include *reference counting* and *generational garbage collectors*.

The execution speed of our SETL persistent runtime system is not entirely satisfactory (see Chapter 7). Our system is slowed down by its interpretive implementation using Scheme, which is itself interpreted. This two-level interpretation is very expensive and should be avoided. There are two possible solutions: compiling SETL programs into object code[10] or interpreting them using compiled programs. The data structures used in the SETL persistent runtime system (e.g., binary search trees) can perform very poorly when data values are large. Better choices include balanced trees and trees with larger fan-out[67]. However, due to the tradeoff between time overhead and space overhead, we face a limit in increasing tree fan-out, beyond which fan-out increases will damage the effectiveness of space usage.

One interesting phenomenon in our persistent runtime system is that the execution stack can be totally eliminated. This would presumably save some memory space. An execution stack is used in normal runtime systems to keep track of the order procedure activation so that execution can return to appropriate place when current procedure activation is complete. This information is redundant in persistent runtime systems, because complete execution history is available so that returns can be made using information available in previous timeslices. Nevertheless, the current implementation does use an execution stack. This makes it easier to implement the system, to compare its performance with normal runtime systems, and to support coarse grained recording.

It may involve little loss, since our experience indicates that a majority of memory usage is spent on recording changes of program variables resident in the runtime heap.

Using more sophisticated selective recording techniques can further improve the performance of our debugger. We know that memory usage of a persistent runtime system can increase proportionally to program execution time in the worst case. This will eventually cause memory overflow problems, which selective recording can avoid in some cases. Selective recording in the current implementation of LSD is supported only by coarse grained recording granularity. Other, more sophisticated possibilities include:

1. Selective recording at the variable, source line, and/or procedure level[65]. In this scheme, persistent runtime systems record execution information only for selected variables, source lines and/or procedures, and discard other information. This scheme resembles the conventional breakpoint method but tends to be more powerful and flexible.
2. Recording only most-recent execution history[20]. In this scheme, persistent runtime systems record only the most-recent execution history (say, the last 2000 execution timeslices) and discard all previous information. This scheme is justified by the assumption that bugs are more likely to be located close to the point at which an error occurs. Although it is hard to characterize the accuracy of this assumption in quantitative terms, it is a useful heuristic in practice.
3. Nonuniformly distributed recording[32]. To extend our current scheme of uniformly distributed recordings, we can distribute runtime state recording nonuniformly. Recording only the most-recent execution history is an extreme case of this scheme. Many other distributions are possible, e.g., a recording scheme with n recordings r_1, r_2, ... r_n, in which program execution is laid out over the interval [0, 1], r_i records runtime state at the execution moments $1 - 1/2^i$, where $1 \le i \le n$.
4. Other options are discussed in recent literature: [56][17].

Although the graphical widgets used in our interface are generally appealing, it can be improved by designing a few custom widgets. One example concerns the three scrollbars

of the history window. A constraint among these scrollbars is that the timeslice of the starting bar is no later than that of the focus bar, which is itself no later than that of the ending bar. Our current implementation (three scrollbars each having one dragging area) does not represent this constraint directly. A better candidate would be a custom scrollbar with three drag areas.

The graphical animation supported by our debugger is confined to a few predefined cases. That is because a more generic animation facility requires more general, sophisticated mappings from variable values to graphical primitives, which often needs additional semantic knowledge at the user level. Thus potential improvement of our current implementation is customized animation, which allows users to add custom viewing procedures. This research area deserves future effort.

A last limitation concerns communication overhead. Implementing our debugger's interface and its runtime system as two separate processes makes each of them relatively easy to design, implement, and debug. However, such an arrangement introduces context switching and communications overhead. This is not a problem of LSD, as its current debugging targets are relatively small. But context switch and communication overhead may become a bottleneck to quick system response when debugging large programs. This limitation must then be alleviated by combining the two components into a single process.

8.2 Summary and Comments on Open Research Issues

The lazy debugging paradigm described in this thesis offers a promising approach to building powerful and high-level debugging tools for a number of programming. Our experience with LSD shows that lazy debuggers will have several major advantages over conventional debugging tools.

1. Lazy debuggers are well adapted to the fundamental requirements of program debugging and easy to use. Users of lazy debuggers do not need to modify their programs. Complete execution histories are available to lazy debuggers so that users

can freely examine large amounts of runtime data objects at arbitrary execution moments. This eliminates a lot of user input, while supporting systematic debugging very comfortably.

2. Debugging a program using our lazy debugger seems to be more efficient than debugging using other debugging tools. Dozens of executions are commonly needed to locate a program bug by conventional means. Lazy debuggers need only one execution. Although the execution uses more time and more memory space than conventional debugging, overhead is generally tolerable. Both the complexity analysis of the lazy debugging paradigm and the actual performance of our lazy debugger prototype indicate that our approach has acceptable time overhead and that its moderate space overhead scales predictably. Furthermore, the single program execution required comprises only a small portion of the complete debugging process. Also, more computer hardware resources will surely be available in the future.
3. Given the complete execution history of a debugging run, lazy debuggers can support many powerful, high-level debugging facilities. The graphical user interface of LSD shows that the innovative visual debugging environments which then become available can help users examine large volumes of runtime objects and detect their relations easily and quickly. Although it is hard to quantify the usability of any visual debugging environment, our experience does indicate that its form of presentation helps users focus on their task rather than on technical ones, and makes the debugging process easier.

Nevertheless, the technology is not so mature that this approach can be applied to every programming language. We now review a few research issues whose answers would improve our lazy debugging paradigm, either by improving its performance, by making it easier to use, or by making it applicable to more programming languages. These issues are:

1. Persistent arrays. Arrays are basic to almost all programming languages. Use of efficient persistent arrays would naturally solve many of the difficulties discussed above.

2. Persistent data structures allowing an unbounded number of access points. Even if no efficient persistent array is available, persistent data structures with an unbounded number of access points would also make the lazy debugging paradigm easily applicable to the many programming languages that support pointer aliasing.
3. Navigation in a large execution information space. Effective means of navigation in a very large information space remains an important research area. Even though our work has developed some techniques for navigating program execution histories, many questions are still open. For example, scrollbars are only effective within some limit, as users often find it very difficult to use then to examine a program consisting of several thousands lines of code. More work needs to be done on strategies for displaying many coordinated program pieces together. The general question of how to present and access large volumes of information in a limited and relatively very small screen space remains critical for many navigation tools, even though some breakthroughs are emerging (e.g., Perlin and Fox's "Pad"[61]).

APPENDIX A References

[1] H. Abelson, G. J. Sussman and J. Sussman, “Structure and Interpretation of Computer Programs”, MIT Press, Cambridge, MA, 1985.

[2] H. Abelson, et al, “Revised^4 Report on the Algorithmic Language Scheme”, 1991.

[3] E. Adams and S. S. Muchnick, “Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations”, *Software - Practice and Experience*, 16(7), 1986.

[4] H. Agrawal, “Toward Automatic Debugging of Computer Program”, Ph.D. Thesis, Purdue University, 1992.

[5] H. Agrawal, R. A. Demillo, and E. H. Spafford, “Debugging with Dynamic Slicing and Backtracking”, *Software - Practice and Experience*, 23(6), 1993.

[6] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers, Principles, Techniques, and Tools”, Addison-Wesley, Reading, MA, 1986.

[7] A. W. Appel, and D. B. MacQueen, “A Standard ML Compiler”, in *Functional Programming and Compiler Architecture*, in G. Kahn (eds.), *LNCS 274*, Springer-Verlag, 1987.

[8] K. Araki, Z. Furukawa, and J. Cheng, “A General Framework for Debugging”, *IEEE Software*, 1991.

[9] M. J. Bach, “The Design of the Unix Operating System”, Prentice Hall, Englewood Cliffs, NJ, 1986.

[10] D. Bacon, private communication, 1993.

[11] R. M. Balzer, “EXDAMS - Extendable Debugging and Monitoring System”, *AFIPS Proceedings of Spring Joint Computer Conference*, 34, AFIPS Press, Montvale, NJ, 1969.

[12] G. Booch, “Object-Oriented Design with Applications”, Benjamin/Cummings, Redwood City, CA, 1991.

[13] J. D. Bovey, “A Debugger for A Graphical Workstation”, *Software - Practice and Experience*, 17(9), 1987.

[14] H.-D. Boecker, and H. Nieper, “Making the Invisible Visible: Tools for Exploratory Programming”, *Proceedings of the First Pan Pacific Computer Conference*, The Australian Computer Society, Melbourne, Australia, 1985.

[15] H.-D. Boecker, G. Fischer, and H. Nieper, “The Enhancement of Understanding through Visual Representations”, *ACM Proceedings of Computer Human Interaction 1986*. ACM Press, 1986.

[16] M. H. Brown, “Algorithm Animation”, MIT Press, Cambridge, MA, 1988.

[17] J. Choi and J. M. Stone, “Balancing Runtime and Replay costs in a Trace-and-Replay System“, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[18] J. Conklinm, “Hypertext: An Introduction and Survey”, *IEEE Computer*, 20(9), 1987.

[19] S. K. Debray, “Static Inference of Modes and Data Dependencies in Logic Programs”, *ACM Transactions of Programming Languages and Systems*, 11(3), 1989.

[20] R. K. Dewar, private communication, 1993.

[21] P. F. Dietz, “Fully Persistent Arrays”, Unpublished draft, 1991.

[22] E. Dijkstra, “Programming Considered as A Human Activity”, Classics in Software Engineering, Yourdon Press, New York, NY 1979.

[23] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, “Making Data Structures Persistent”, *Journal of Computer and System Sciences*, 38, 1989.

[24] M. Eisenstadt, “Tales of Debugging from the Front Lines”, in J. Spohrer and C. Cook (Eds), *Empirical Studies of Programmers: Proceedings of the Fifth Workshop*, Norwood, NJ, 1993.

[25] P. M. Ferguson, “The Motif Reference Manual”, Volume 6B of the O’Reilly Series on X, O’Reilly Associates, 1983.

[26] D. P. Friedman, C. T. Haynes, and E. Kohlbecker, “Programming with Continuations”, in P. Pepper (eds.), *Program Transformation and Programming Environments*, Springer-Verlag, 1984.

[27] T. Fukaya and M. Nagata, “An Automatic Debugging Approach for Logic Programming with a Method for Propagating Constraints”, *COMPSAC’91, The 15th Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, 1991.

[28] J. Griffin, “Parallel Debugging System User’s Guide”, Technical Report, Los Alamos National Laboratory, 1987.

[29] R. E. Griswold, “The Implementation of the Icon Programming Language”, Princeton University Press, Princeton, NJ, 1986.

[30] R. E. Griswold and M. T. Griswold, “The Icon Programming Language”, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[31] P. K. Harter Jr., D. M. Heimbigner, and R. King, “IDD: An Interactive Distributed Debugger”, *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, 1985.

[32] J. Hong, private communication, 1993.

[33] IEEE, “IEEE Standard Glossary of Software Engineering Terminology”, IEEE, New York, NY, 1990.

[34] S. Isoda, S. Takao and O. Yuji, “VIPS: A Visual Debugger”, *IEEE Software*, 4(3), 1987.

[35] D. R. Jefferson, “Virtual Time”, *ACM Transactions on Programming Languages and Systems*, 7(3), 1985.

[36] G. W. Jones, “Software Engineering”, John Wiley and Sons, New York, NY, 1990.

[37] B. Korel and J. Laski, “Dynamic Program Slicing”, *Information Processing Letters*, 29(3), 1988.

[38] L. Lamport, “Time, Clocks, and the Ordering of Events in a distributed System”, *Communication of ACM*, 21(7), 1978.

[39] S. Lauesen, “Debugging Techniques”, *Software - Practice and Experience*, 9(1), 1979.

[40] T. J. LeBlanc and J. M. Mellor-Crummey, “Debugging Parallel Programs with Instant Replay”, *IEEE Transactions on Computer*, 36(4), 1987.

[41] C. H. LeDous, and D. S. Parker Jr., “Saving Traces for Ada Debugging”, *Ada in Use, Proceedings of the Ada International Conference*, ACM, Cambridge University Press, 1985.

[42] M. A. Linton, “The Evolution of Dbx”, *Proceedings of the 1990 Usenix Summer Conference*, Anaheim, CA, 1990.

[43] Z. Liu, “Computer Support for Information Exploration and Presentation: A Survey”, Department of Computer Science, New York University, 1992.

[44] J. Lyle, “Evaluating Variations on Program Slicing for Debugging”, Ph.D. Thesis, University of Maryland, 1984.

[45] A. Di Maio, S. Ceri and S. C. Reghizzi, “Execution Monitoring and Debugging Tool for Ada using Relational Algebra”, *Ada in Use, Proceedings of the Ada International Conference*, ACM, Cambridge University Press, 1985.

[46] C. E. McDowell, and D. P. Helmbold, “Debugging Concurrent Programs”, *ACM Computing Surveys*, 21(4), 1989.

[47] R. Milner, M. Tofte, and R. Harper, "The Definition of Standard ML", MIT Press, Cambridge, MA, 1990.

[48] T. G. Moher, "PROVIDE: A Process Visualization and Debugging Environment", *IEEE Transaction on Software Engineering*, 14(6), 1988.

[49] B. A. Myers, "Displaying Data Structures for Interactive Debugging", Xerox PARC Technical Report CSL-80-7, 1980.

[50] B. A. Myers, "Incense: A System for Displaying Data Structures", *SIGGRAPH'83 Conference Proceedings of Computer Graphics*, 17(3), 1983.

[51] B. A. Myers, R. Chandhok, and A. Sareen, "Automatic Data Visualization for Novice Pascal Programmers", *1988 IEEE Workshop on Visual Languages*, 1988.

[52] B. A. Myers, "The State of the Art in Visual Programming and program Visualization", in Alistair Kilgour and Rae Earnshaw (Eds.), *Graphics Tools for Software Engineers*, Cambridge University Press, Cambridge, Great Britain, 1989.

[53] G. J. Myers, "The Art of Software Testing", Wiley, New York, NY, 1979.

[54] T. H. Nelson, "Interactive Systems and the Design of Virtuality", *Creative Computing*, 6(11, 12), 1980.

[55] T. H. Nelson, "Computer Lib/Dream Machines", Rev. ed., Tempus Books of Microsoft Press, Redmond, WA, 1987.

[56] R. H. B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs", *ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, CA, 1993.

[57] J. Nielson, "Hypertext and Hypermedia", Academic Press, San Diego, CA, 1990.

[58] J. Nielson, "Usability Engineering", Academic Press, San Diego, CA, 1993.

[59] D. A. Norman, "The Design of Everyday Things", Doubleday, New York, NY, 1990.

[60] Panel Session on Debugging Methodology, *Proceedings of Symposium on High Level Debugging*, Asilomar, CA, 1983.

[61] K. Perlin and D. Fox, "Pad: An Alternative Approach to the Computer Interface", *Proceedings of 1993 ACM SIGGRAPH Conference*, 1993.

[62] D. Plattner and J. Nievergelt, “Monitoring Program Execution: A Survey”, *IEEE Computer*, 1981.

[63] C. Rutkowski, “An Introduction to the Human Applications Standard Computer Interface, Part 1: Theory and principles”, *BYTE*, 7(11), 1982.

[64] D. Savic, “Object-Oriented Programming with Smalltalk/V”, Ellis Horwood, 1990.

[65] E. Schonberg, private communication, 1993.

[66] J. T. Schwartz, “An overview of Bugs”, in R. Rustin (eds.), *Debugging Techniques in Large Systems, 1st Courant Computer Science Composium*, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[67] J. T. Schwartz, R. K. Dewar, E. Dubinsky, and E. Schonberg, “Programming with Sets: A Introduction to SETL”, Springer-Verlag, 1986.

[68] J. T. Schwartz, “Proposal for Systematic Debugging Technique Using Inductive Assertions”, Technical Report #320, Computer Science Department, NYU, 1987.

[69] E. Y. Shapiro, “Algorithmic Program Debugging”, MIT Press, 1983.

[70] T. Shimomura, and S. Isoda, “VIPS: A Visual Debugger For List Structures”, *COMPSAC’90, The 14th Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, Chicago, IL, 1990.

[71] T. Shimomura, and S. Isoda, “CHASE: A Bug-Locating Assistant System”, *COMPSAC’91, The 15th Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, 1991.

[72] B. Shneiderman, “Reflections on Authoring, Editing, and Managing Hypertext”, in Edward Barrett (eds.), *The Society of Text*, MIT Press, Cambridge, MA, 1989.

[73] B. Shneiderman, “Design the User Interface: strategies for effective human-computer interaction”, 2nd edition, Addison-Wesley, Reading, MA, 1992.

[74] R. Snodgrass, “Monitoring in A Software Development Environment: A Relational Approach”, *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, ACM Press, 1984.

[75] R. M. Stallman and R. H. Pesch, “A Guide to the GNU Source-Level Debuggers”, Free Software Foundation, 4.01 revision, 2.77 edition, 1992.

[76] Sun Microsystems, Inc., “OLIT 3.0 Widget Set Reference Manual”, Sun Microsystems, Inc., 1991.

[77] R. E. Tarjan, “Amortized computational complexity”, *SIAM Journal on Algebra and Discrete Methods*, 6(2), 1985.

[78] M. Timmerman, F. Gielen and P. Lambrix, “High Level Tools for the Debugging of Real-Time Multiprocessor Systems”, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[79] F. Tip, “A Survey of Program Slicing Techniques”, to be appeared in *Computing Survey*, 1994.

[80] A. P. Tolmach and A. W. Appel, “Debugging Standard ML Without Reverse Engineering”, *Proceedings of 1990 ACM Conference on Lisp and Functional Programming*, ACM Press, 1990.

[81] P. R. Wilson, “Uniprocessor Garbage Collection Techniques”, Submitted to *ACM Computing Surveys*, 1993.

[82] P. R. Wilson and T. G. Moher, “Demonic Memory for Process Histories”, *Proceeding of SIGPLAN 89 Conference on Programming Language Design and Implementation*, 1989.

APPENDIX B Test Programs

B.1 SETL Quadruple Scanner

```
program scanner;
-- This program scans a SETL quadruple file generated by David Bacon's SETL translator,
-- strips any unnecessary code in the file, and prints its quadruples.

   read (file);
   open (file, "TEXT");
   geta (file, line);
   while not eof loop
      if line(1) /= "#" then
         if line(1) = "%" then
            context := line(2..);
         elseif context = "CODE" then
            print (line);
         end if;
      end if;
      geta (file, line);
   end loop;

end scanner;
```

B.2 Prime Numbers

```
program primes;
-- This program computes all the prime numbers up to N,
-- using the sieve of Eratosthenes.

   read (n);
```

```
    primes := [2];
    candidates := {3,5..N};

    for num in [3,5..N] | num in candidates loop
        primes with := num;
        candidates less := num;
        for multiple in [num*num,(num+2)*num..N] loop
            candidates less := multiple;
        end loop;
    end loop;

    print ('Primes in the range 1 to',N,':');
    print (primes);

end primes;
```

B.3 All-Solution N-Queens

```
program N_Queen;
-- This program computes all the solutions of the N-queens problem,
-- for an arbitrary natural number N.
-- Thank Zhijun Liu for this program.

    var n;
    read (n);
    solve (1,[[1..n]]*n,[]);

    procedure solve (index, tab, answer);

        var newtab := [];
        if index=n+1 then
            print (answer);
        else
            for i in tab (index) loop
                for tab_index in [index+1..n] loop
                    newtab (tab_index) := [x:x in tab (tab_index) | x/=i
                                           and abs (x-i) /= abs (tab_index-index)];
                end loop;
                solve (index+1, newtab, answer+[i]);
            end loop;
        end if;

    end solve;

end N_Queen;
```