Chapter 1

Introduction

Ensuring program correctness is an essential but difficult task in software development. Usually, there are two approaches to achieve this task. In the first approach, programs are assumed to contain errors, which are usually referred to as ***bugs***. These programs are executed and their behavior is monitored. If any erroneous behavior is found, the bug should be located and corrected. This activity is known as ***debugging***. In the second approach, requirements of a program are formally specified. Then, this program is derived directly from these given specifications. Alternatively, the program is separately written, and its correctness is verified against the specifications.

Despite the attractiveness of the program derivation and verification approach where program correctness is "guaranteed" by formal methods, debugging is and always will be no less important than any other approach. First, state-of-the-art formal specification techniques are still very limited, and are either impossible or impractical to be applied to many programs [DeMi79, Fetz88, Loec84]. Furthermore, because of the very nature of human beings to make mistakes, even with perfect derivation and verification tools, errors are still inevitable. For example, the specification itself may be wrong, or mistakes may possibly be made when the supposedly correct specification is entered into the computer. In other words, as long as human effort is involved, debugging is still necessary, although it might be at a different level, i.e., debugging the specification.

In this dissertation, we focus on issues of debugging techniques and implementation of tools that assist programmers to locate bugs in their programs so that they can be fixed. We further show that the techniques and tools described can also be applied to solve a variety of related problems.

This dissertation explores the difficulties in debugging sequential and concurrent programs, proposes new approaches for debugging these programs, describes new techniques for improving the implementation of debugging tools, outlines a methodology for building debuggers in general, and finally suggests how these new approaches and implementation techniques can be generalized and applied to the construction of other related tools. For each of the new ideas proposed, prototypical programs were implemented to demonstrate its feasibility.

1.1. Overview of Debugging

Over the years, many software tools have been developed to aid programmers in debugging sequential programs (e.g., [ACM83, Adam86, Kats79, Stal91b]). These ***debuggers*** typically allow programmers to suspend the execution of a given program, examine in detail or modify its states, and then resume its execution. Using a debugger, a programmer can obtain much valuable information about the behavior of the given program and gradually narrow down the possible locations of the errors. However, most of these debugging tools are still unsatisfactory.

They usually require the programmer to control the execution of the program by giving unnecessarily detailed, explicit instructions; they also present the collected information at a very low-level of abstraction (e.g., line number in the source program, values of variable, etc.) instead of at the level of abstraction present in the program. Very little previous work has explored further how the debugger can present this information at a higher level of abstraction.

In contrast to debugging sequential programs, debugging concurrent programs has not received much attention until the last decade. ***Concurrent*** programs differ from sequential programs in that they contain multiple processes executing at the same time. These processes cooperate with each other and together they work toward a common goal. Concurrent programs that have all their processes running on the same machine and communicating through shared objects are called ***parallel*** programs. A typical parallel program runs on a shared-memory based multiprocessor machine. It may also run pseudo-concurrently on a single processor machine that supports multitasking. On the other hand, a ***distributed*** program is a concurrent program in which the processes reside on different physical machines and communicate through message passing. As high performance multiprocessor machines and high speed computer networks become more and more common, many large scale concurrent programs have been built (e.g., [Acce86, Allm83, Cher88, Dubo82, Hell78, Hume87]). By utilizing the combined power of multiple processors, they can shorten the execution time of many programs and can make them more tolerant to faults. However, concurrent programs are usually much more complex than sequential programs due to the presence of multiple processes that interact with each other. Programming these interactions is often prone to errors. Furthermore, these processes execute independently with different speeds, and the relative ordering of their executions is non-deterministic. As a result, running the same program twice with the same input might produce different results. All these properties make error detection and location in concurrent programs much more difficult than in ordinary sequential programs. Existing techniques and tools for debugging sequential programs have proved to be inadequate for debugging concurrent programs. Over the last few years many new approaches have been proposed for debugging concurrent programs. They are outlined in the next chapter. However, we are not satisfied with these approaches and thus decided to develop better solutions, which are described in this dissertation.

1.2. Key Issues in Debugging

Debugging is a complex task and many problems need to be solved. It is a task that involves heavy interactions between the ***programmer*** and the ***inferior process***. By *programmer* (or *user*), we refer to the person carrying out the task of debugging, whereas an inferior process (or simply *inferior*) is the executing process (or processes) that is being debugged. There are four major issues in debugging:

- *Gathering information*—To effectively locate and diagnose any error in an inferior process, a programmer needs to collect as much information as possible about the execution behavior of the inferior process.
- *Presenting information*—The amount of information collected might be very large and incomprehensible for human beings. Therefore, tools must be made to "intelligently" filter out the irrelevant information and present to the programmer only that information that might lead to correct diagnosis of the problem.

- *Interacting with programmers*—Since these debugging tools perform a very complex task, they must be easy enough to use so that any programmer can specify exactly what sort of information he or she wants.
- *Implementation*—The implementation of debugging tools is by itself a complicated software engineering problem that deserves special attention. Implementing experimental debuggers is a very tedious and time-consuming job. It is therefore desirable to study how debuggers can be designed so that most of the code can be shared by different implementations.

1.2.1. Gathering Information

One basic question is how information about the behavior of the inferior process can be obtained. One approach is to modify the inferior's source program and insert debugging statements to output relevant information such as names of procedures being called or values of variables. However, this approach suffers from having to recompile the source program every time debugging statements are inserted. Besides being very costly and time-consuming, it interrupts the debugging session and distracts the programmer's attention. A more common approach is to use a separate process to "peek" into the inferior process's address space and grab whatever information that is required. This approach usually slows down the execution of the inferior: In single-processor machines, the inferior has to be stopped while its memory content is being searched by the debugging process. In multi-processor environments, memory contention may occur if the debugging and inferior process both read the same memory location at the same time.

A well-known problem that is particular to distributed programs is the ***global state*** problem [Lamp78]. Information collected from various inferior processes has to be transferred through the communication network back to the debugging process or the programmer, and because of the delay induced by the network, this information reflects only the behavior of a process at some point backward in time. In other words, analysis based on the information received could potentially be incorrect because the exact behavior of that particular process might have been changed.

Another important issue is that the very act of information gathering might already be enough to alter the behavior of an inferior process, and thus cover up problems that might otherwise appear. For example, insertion of debugging statements changes the position of the code in the inferior, and the presence of a debugging process also slows down the execution of the inferior process. In the case of concurrent programs, the relative order of execution of individual thread might also be changed. Therefore, it is very important that a debugger should keep its intrusiveness to the inferior to a minimum.

1.2.2. Presenting Information

Before the exact location of an error is found, a programmer usually would like to collect as much information about the inferior's execution as possible so that erroneous behavior will not go undetected. However, the amount of information collected could be overwhelming and thus incomprehensible by human beings. Any debugging aid should therefore selectively present this information to the programmers, namely, only the subset of information that might help diagnose the problem. Obviously, the issue here is how programmers decide which pieces of information is irrelevant and which is not, and how raw data can be turned into concise, meaningful information.

The mechanism for extracting useful information can be passive or active. A passive mechanism applies analysis solely on the collected information, possibly filters out the irrelevant parts, and presents the remainder to the programmer. This approach allows efficient information extraction with minimal influence on the inferior's execution. But the kind of information to be extracted is predetermined and cannot be changed after execution has begun.

On the other hand, an active mechanism carries out analysis while information is being collected, and might generate feedback to adjust the kind of information to be collected. Carrying out analysis "on-the-fly" requires extra overhead and might slow down the execution of the inferior, but this method is very flexible and can dynamically adapt to the changing behavior of the inferior process.

When raw data have been analyzed, digested, and condensed into useful information, the debugger tools have to present it in some easily comprehensible way to the programmer. This can be expressed in terms of text, figures, graphs [Soch88, Zern91], or even sound [Fran91]. Whatever the form of expression is, the main purpose is to give programmers an insight into the behavior of the inferior process. It is therefore important to study how information can be presented at a level of abstraction close to that in the inferior programs. Too much detail might distract a programmer's attention, while over-abstracted expressions might not give sufficient information to be useful.

1.2.3. Interacting with Programmers

Although debuggers can be equipped with many powerful functions to collect information and analyze the behavior of the inferior process, ultimately it is the programmer who is doing the job of debugging. A programmer must decide what kind of information should be collected, how this information should be analyzed, and what level of abstraction should be used when results are presented. In other words, a debugger must provide ways for programmers to specify the actions it should carry out.

The problem is how such an interface between programmers and debuggers should be designed. A debugger should be powerful enough to carry out all kinds of operations that a programmer might want it to perform. Furthermore, the interface should be simple and easy to use so that operations can be specified with minimal effort. Typical debuggers provide some form of command language for controlling their operations. Designing a command language that is both powerful and expressive is a major challenge.

The structure of a debugger command language is closely related both to the information collection and abstraction mechanism provided by the debugger, as well as to the language in which the inferior program is written, i.e., the ***target language***. A command language that closely resembles the target language allows programmers to specify operations in a way similar to how they write programs. On the other hand, if the same debugger or debugging mechanism is used to debug programs written in different target languages, a command language that is independent of these target languages and tailored for the debugging mechanism would be more desirable. Since generality and specialization usually cannot be achieved at the same time, a good compromise has to be found.

1.2.4. Implementing a Debugger

The implementation of debugging tools involves many important issues, mainly because debuggers are very complex programs by themselves. Implementing debuggers for interpreted programs is relatively less complex, because most of the necessary support for debugging can be incorporated into the language interpreter. The overhead required in supporting debugging is relatively low, when compared with the cost of interpreter. On the other hand, implementing debuggers for compiled programs is especially hard, because of the many constraints imposed by the limitations of the hardware and operating system.

For debugging compiled programs, operating system support is essential. At a minimum, an operating system must provide mechanisms for debuggers to examine or modify the address space of inferior processes—an operation that directly violates the ordinary function of multi-programming operating systems where an individual process's address space is protected from access by others. Many existing operating systems were designed with minimal consideration taken for debugging support. Their mechanisms for debuggers to monitor and control inferior processes are poor and inefficient. Very often the interactions between a debugger and the inferior process through the operating system is the bottleneck in the debugger's performance. It is therefore important to study the kind of debugging support operating systems should provide, and how they can be implemented efficiently.

Architectural support is also important. Some operations performed by a debugger are very expensive if implemented in software. A typical example is to find out when the contents of a variable is modified. Identifying those debugging operations that would best be performed in hardware is a significant issue. Such operations should be implemented in hardware. For existing hardware that does not support such operations, we should be careful in using these costly operations.

Given that existing machines and operating systems do not provide ideal support for debugging, the question is then, How a debugger can be implemented within the given constraints? We need to develop techniques for implementing desired debugging operations with maximum efficiency and minimum influence to the normal execution of the inferior process.

Another issue in building debuggers is how to implement debuggers that are portable to as many different operating systems and hardware as possible, and how to make them capable to debug programs written in as many different target languages as possible. Since a debugger is such a complex program by itself, reimplementing a debugger for every different environment is undesirable. Designing a portable debugger is not a simple task because many parts of a debugger depend heavily on the specifics of the target language, the operating system, as well as the machine architectures. How to structure a debugger so that the portable parts can be separated from the non-portable parts, and develop a standardized framework for putting these parts together is an important design issue.

1.3. Overview of Proposed Solutions

Most of the problems of debugging described in the last section will be discussed in this dissertation. We propose several new solutions and have implemented prototypes of some of these solutions to demonstrate feasibility. These solutions are outlined in this section, and they will be discussed in details in subsequent chapters.

The first design decision that we make is not to modify any part of the source program for debugging purposes. Besides being costly in both time and resources, inserting debugging code and re-compiling a program might affect its behavior and can cover up old bugs or introduce new ones. We use a separately executing debugger to collect information about the execution behavior of inferior processes. This debugger controls the execution of the inferior processes, and directly reads or writes to the inferiors' address spaces. The existence of a debugger inevitably affects the behavior of the inferiors, but new techniques are proposed to minimize such effects.

Many existing debuggers access the address space of the inferior process through a special system service. Typically, a debugger controls the inferior's execution by inserting a trap instruction into the inferior's address space. Upon executing this trap instruction, the inferior is stopped, and the debugger is notified. The debugger can then examine or modify the inferior's memory content, and finally resume the inferior's execution. This method is very inefficient because it requires quite a number of context switches. The technique that we propose can eliminate most of such context switches. Instructions for examining the execution state and collecting information are injected by the debugger into the inferior's address space. Instead of passing control back and forth between the inferior and the debugger, execution is redirected to the injected instructions. Thus, most of the debugging operations are carried out within the inferior's address space and the same thread of control and the debugger is called into action only when necessary. If the underlying operating system supports shared memory, a debugger can make the inferior's address space sharable and can then directly access any memory location. Similar approaches are used by [Aral88b, Kess90].

Injecting debugging code to the inferior process, or removing code afterwards, requires the ability to dynamically link or unlink such code to the inferior's address space. We have developed a dynamic link/unlink editor called *dld* that solves this problem. The ability to link and unlink object files while a process is executing actually has more applications other than just debugging. It is a new approach to program construction where the functionality provided by a compiled program during its execution can change with time or in response to the environment. Thus, this approach retains the efficiency of executing native machine code and adds the flexibility of modifying a program during its execution.

In terms of the mechanism for selecting and filtering useful information and presenting it to the programmer at the right level of abstraction, we developed a ***dataflow approach*** to ***event-based*** debugging. The notion of event-based debugging has received much attention in the last few years [Baia86, Bate88, Elsh88, LeBl85]. It involves the recognition and analysis of the occurrences of interesting activities (*events*) of the program. A programmer then tries to deduce from the results of the analysis the source of the programming errors. Many studies have been devoted to applying this technique to distributed debugging. A major concern in using this method is that the number of events is usually very large, and some filtering mechanism must be used to limit the amount of information presented to the programmers. To solve this problem, we developed the dataflow mechanism.

In the dataflow approach, we employ a (coarse-grained) dataflow view of events: a high-level event is recognized when an appropriate combination of lower-level events on which it *depends* has occurred. Event recognition is controlled by familiar programming language constructs, such as conditional and repetitive statements and user-defined functions. This dataflow approach, therefore, permits "active" event recognition. It allows arbitrary actions, specified in the debugger language, to occur when attempting to form higher-level events. It also permits actions to occur on "near misses", e.g., when some but not all of the desired lower-level events have occurred or when information gleaned from occurrences of lower-level events "nearly" satisfies a desired condition. This dataflow approach is, therefore, more flexible and powerful than existing pattern matching schemes for event recognition. It is also more accessible to the user: the way that users specify event recognition is close to the way that users write programs. We have implemented a prototypical debugger called *Dalek*[1] for sequential programs that employs this dataflow approach. Details of the design and implementation, as well as experience using Dalek are discussed in Chapter 2.

On the implementation side, we attempt to address a more general problem and consider not only the implementation of traditional debuggers, but also that of monitoring tools, such as instruction counters and program profilers. We categorize the basic functionality required by all these tools, and design and implement a layered model for them. Under this model, the interfaces between adjacent layers are well-defined so that different implementations of the same layer can be interchanged transparently to other layers. Such flexibility is very important for building experimental debuggers or monitors, as well as for maintaining existing tools. A standardized model is also important in distributed environments where programs might be running on heterogeneous systems. By hiding

[1]A Dalek is a semi-mechanical alien life form that was opposed by Doctor Who in its efforts to "exterminate" everything in sight [Tull83].

the differences of different systems in the lower layers and standardizing their interface to the upper layers, the same higher-level *front end* can run on any system without modification. We believe that the model proposed can provide a generalized, system independent platform on top of which different debugging and monitoring tools can be built. In Chapter 6, we describe the implementation of some of the prototypical debuggers that we have built using this model, and the experience we gained from them.

1.4. Summary of Contributions

One significant contribution of this dissertation is the design of the dataflow approach to event-based debugging, as well as the idea of a fully programmable interface to control the debugger. A prototype debugger to illustrate their advantage has also been implemented. The design and implementation of a dynamic link/unlink editor is also an important contribution. In fact, our implementation of this design has already widely been used in many application programs. Furthermore, the proposed layered model for building debuggers, and its generalization to other monitoring tools in general will considerably help establish a standard for the implementation of portable and easily maintainable debugging and monitoring tools.

1.5. Organization of this Dissertation

The rest of this dissertation is organized as followed. We begin in Chapter 2 with a general discussion on the problems in debugging sequential and concurrent programs, and a summary of previously proposed approaches and techniques to solve these problems. In Chapter 3, we present a new, dataflow approach to debugging.

We also describe the prototype debugger we built that applies this dataflow approach, and give examples showing how this debugger is used to attack difficult debugging problems. In Chapter 4, we discuss an entirely new concept of dynamic linking, which, as described in later chapters, not only makes possible many new, advanced debugging techniques, but also has numerous applications in many different areas. In Chapter 5, we focus on technical issues in the actual implementation of debuggers. We describe the basic components of a typical debugger, the technical problems and constraints that make their implementations difficult, and how these problems can be solved. In Chapter 6, we propose a novel, layered approach that standardized the constructions of debuggers. This layered model provides guidelines for implementing debugger in an organized way. It also helps making debugger more portable across machines with different architectures and operating systems, and allows many other monitoring tools to be built quickly and easily. In Chapter 7, we describe the possible directions to which debugging techniques can be improved, and other interesting work that can be done but has not been included in this dissertation.

Chapter 2

Background

In this chapter, we describe previously proposed approaches to software debugging. We also discuss problems that are particular to debugging sequential, parallel, and distributed programs. Lastly, we summarize some of the recent work on debugging and describe their approaches to solve these problems. Background knowledge about the implementation of a debugger is described in Chapter 5.

Perhaps the reason why debugging is so hard is that it is a process of searching for the unknown. Usually, when a programmer begins to debug a program, the only information available is that the program is not doing what it is supposed to do. Many errors in programs are very subtle and it is not obvious what these errors are and where they are located. Thus, programmers do not usually know exactly for what they are looking when they start debugging a program. Furthermore, it is very hard to define what an error really is until it has been found. Because of this fuzzy nature of errors, it is very difficult to build tools that automatically locate errors *for* the programmers, despite some success in a very limited scope [Shap83]. As a consequence, most debuggers are actually tools that assist collecting information about the execution behavior of programs, and the actual job of identifying and locating errors is left to the programmers.

One of the most primitive methods of debugging is to capture the complete memory image of the executing process and record it onto the printer or disk. This method has been used widely for years, and nowadays many operating systems still perform this kind of "core dump" automatically upon detection of errors such as division by zero or invalid memory address reference. Under this approach, an executing program is viewed as a finite state machine and its ***state*** at any given time instant is represented by its memory image. By examining and analyzing a snapshot of this image, a programmer can derive useful information about the execution of the program, which may lead to locating the source of the errors.

One drawback of the state examination approach is that it only gives static information that can be derived from the memory image, which is merely a snapshot of the program execution. It tells a programmer *what* happened at that time instant, but reflects nothing on *how* the process arrived at that state, or through what other states the program had gone to get to that state.

The *behavioral approach*, on the other hand, observes a program's activities dynamically during its execution. This kind of observation is commonly obtained by inserting output statements into the source program to print out relevant information during the program's execution. A program is considered correct only if it *behaves* as expected during its entire execution. If any abnormal activity is detected, the process's execution history is examined and the cause of the error might then be found. In addition to output statements, assertion statements might also be used. These statements are short pieces of code embedded into the source program to verify that the relationships among the values of variables are correct.

Over the years, many tools have been developed to assist programmers in debugging. Usually they employ both the state examination and behavioral approaches. Many of these debuggers (e.g., *dbx* [Adam86, DBX83], *sdb* [Kats79], *gdb* [Stal91b], *DEBUG* [DEC88], and the Turbo system debugger [Borl85]) allow programmers to suspend a process in the middle of its execution, examine or even modify its state, print out selected information that might be related to the errors, and then resume the process's execution. *Source-level* debuggers can even display the program state at the level of the source programming language, thus helping programmers to correlate the state with the program sources. Some debuggers provide facilities to trace the program's execution and print out its activities (such as invocation of procedure or execution of any specified statements) automatically. These facilities are effectively an automated mechanism for inserting debugging output statements without modifying the source program.

2.1. Debugging Sequential Programs

One characteristic of a sequential program is that it is very easy to obtain a snapshot of the complete state of the process executing the program. Since the state is easy to obtain, most of the existing sequential debuggers apply the state examination approach. Note that the state examination approach is feasible for sequential programs because, except for some real-time programs, the speed of execution in general does not affect the results at all. In other words, one can suspend a process from execution, examine its memory content and then resume its execution without affecting the outcome of the program. Furthermore, given the same input, a sequential program always goes through the identical states in its execution. Hence, the same error will show up when the program is run for the second or subsequent time. This property is known as ***deterministic***.

State-of-the-art sequential debuggers, such as *dbx* [Adam86, DBX83], *sdb* [Kats79], *gdb* [Stal91b], and VMS's *DEBUG* [DEC88], usually allow programmers to control the execution of the inferior process interactively. Besides stopping the inferior process to examine its memory contents, these debuggers can also be used to modify its state or change the path of its execution. Programmers can then either skip over some unimportant functions or repeatedly execute parts of the program that are thought to be erroneous. Source level debuggers provide a better interface by creating the illusion that the machine is directly executing the program at the source language level. Execution of the program can be thought of as occurring at the source level. Through these debuggers, programmers can specify locations in the program by line number or function names, and refer to variables by the same names used in the source program.

Normally, debugging of sequential programs is a cyclic activity. When abnormal behavior is observed, the state of the inferior process is examined in order to find out which part of the program is currently being executed. Then the program is executed again from the beginning and stopped at some distance before the last location where the error was found. The programmer can then control the inferior process to go one small step (usually one source line) at a time, and observe how its state changes. The whole process is repeated until the exact location that causes the error is found. This method assumes that the inferior's behavior is deterministic.

As described earlier, the state examination approach has the disadvantage of not being able to capture the execution history of the inferior process. Usually this approach can be complemented by putting breakpoints at various locations in the program and making the debugger print out information about the current activities of the inferior at that point in time. However, no other work has gone beyond this to give a deeper study on applying the behavioral approach to sequential debugging. We believe that a debugger should present the information about the activities of the inferior process not only at the source language level, but also at any higher level of abstractions as desired by the programmers. Furthermore, even for those few debuggers [DBX83] that provide automatic traces of function invocations or execution of specified statements, their implementation methods are very costly and cause the execution speed of the inferior to slow down by orders of magnitude, thus making them impractical.

2.2. Debugging Parallel Programs

As stated in Section 1.1, a parallel program is a concurrent program that has all its processes executing on the same machine. This definition includes those concurrent programs that execute under a single-processor multi-task operating system, where there is pseudo-concurrency, or those that execute on shared-memory based multiple processors machines, where there is true concurrency. Processes of a parallel program execute independently of each other—they do not necessarily execute the same machine instructions, or even if they do, their relative execution speed might be different. Therefore, even though a correct parallel program should always achieve the specified goal correctly, the interleaving of any point in the execution of the corresponding processes is non-deterministic. Thus, parallel programs have the property that, during their execution on a given set of inputs, the behavior of the same program might be different.

Different processes of a parallel program exchange information by means of shared objects, usually in the form of shared memory. Because of the difference in relative speed, and in order to ensure correct communication, these processes sometimes need to synchronize with others by blocking themselves temporarily. Through such synchronization and the exchange of information, the flow of control of any particular process can be affected by the execution of others.

Due to these characteristics, many of the debugging techniques for sequential programs cannot be applied directly to parallel debugging. The non-deterministic behavior makes it almost impossible to use cyclic debugging, unless some special mechanism is imposed to guarantee that re-executing the program will lead to the same previous states. Therefore, applying the state examination approach means that all information must be collected in a single run, because intermediate states of the process among different runs might be inconsistent and cannot be combined together for analysis. However, information collected from a single run is usually inadequate. Beside non-determinism, the necessary process synchronization also creates many difficult debugging problems. Process synchronization is a very tricky and error prone part of a parallel program. Many incorrect solutions are not obvious and their flaws only show up under very specific conditions. For example, when all processes block themselves and wait for some conditions that will never happen, the program will stop forever. This case is called a *deadlock*. On the other hand, when some of the processes do not stop when they should, they might access the shared objects at the wrong time, corrupt the shared data, and hence interfere with the normal execution of other processes. These errors can be very subtle and might show up only when the processes interact in a particular order that rarely happens. Such errors are very difficult to detect and reproduce. In the parallel environment, the intrusiveness of the debugger becomes significant, too. That is, single-stepping any of the processes, or merely the presence of the debugger might affect the relative speed of the processes, and thus cause bugs to disappear (or new ones to appear) when the program runs under debugger's control.

The non-deterministic characteristics of parallel programs has lead to the approach of ***monitoring and replay*** [Aral88b, LeBl87, Mill88] in parallel debugging. Since the exact execution pattern cannot be reproduced, the activities of the program are monitored and recorded during its execution. This recorded information is replayed later for analysis. Enough information must be collected to ensure that everything related to the error is recorded, or can be derived from the records. This execution replay gives an outline of the process's behavior with respect to the recorded activities. It can be carried out any number of times for analysis and a programmer can gradually narrow down his or her focus until the exact location of the bug is found. An enhancement of this approach applies recording and *controlled re-execution.* Instead of merely replaying what has been recorded, the debugger records the interleaving of the processes and then re-executes the program again. But this time it forces the program to execute according to the recorded ordering so that the interleaving of the processes is the same as that in the first run. Thus, the behavior of the program can be regenerated. Since the program is actually being executed again, no information is lost and programmers are free to examine the states of the program in arbitrary detail. Note that cyclic debugging can then be used, because now the relative ordering is fixed.

2.3. Debugging Distributed Programs

Unlike parallel programs, which run on a single machine, possibly with multiple processors, ***distributed*** programs consist of multiple processes running on multiple machines connected by a computer network. They share many of the characteristics of parallel programs, such as non-determinism, existence of multiple processes that interact with each other through exchange of information, and the need to synchronize these processes in order to ensure correct execution. The major difference between the parallel and distributed programs is the presence of the computer network and, therefore, how processes communicate. Processes in parallel programs communicate through shared objects, while in the distributed environment, the only connection between any two machines is the data transmission link. Otherwise there is no shared object. Thus, in order to pass information from one process to another that resides on a different machine, it must be put into *messages* and then sent out through the network.

A consequence of communication via the network is that the time taken to deliver messages from one machine to another is much longer than that of accessing memory. Also, the exact delay in any particular message transmission is unpredictable, which results in another source of non-determinacy. Furthermore, processes on different machines are executing in different environments. The work load of the machine, processing speed, and even the value of the real-time clock are all different. Therefore, message passing is the only way to obtain the state of other machines. And because of the network delay, by the time information about the state of one machine arrives at other machines, its state has already changed. Therefore, it is impossible to obtain the *global* state of the whole program unless all processes are suspended from execution. In fact, even the relative ordering of activities in different processes is hard to obtain, because each machine has its own clock and there is no guarantee that these clocks agree with each other. Simply obtaining the relative ordering requires specially designed mechanisms [Fidg88, Lamp78].

The characteristics of distributed programs make debugging them extremely hard. Like parallel programs, their execution behavior is non-deterministic and non-reproducible. In a shared-memory multiprocessor machine, a snapshot can be taken by stopping all processes at once and then recording the shared memory contents. But in the distributed environment, the notion of global state does not even exist. It is not possible to get an instantaneous snapshot of a distributed program because the snapshot command itself takes time to travel through the network. The best that can be done is to broadcast a message to tell all processes to stop first; then the snapshot can be taken [Chan85, Spez86]. Stopping a distributed program is not a trivial task at all. Care must be taken to ensure that all messages that are being transmitted on the network are properly received before allowing the recipients to stop. Otherwise, messages might be lost. The complexity of obtaining a snapshot encourages the use of the monitoring and replay approach. One advantage of this approach is that after recording the activities of the program, its exact behavior can be replayed or reproduced correctly. However, in the distributed environment, it is very difficult to determine the order of occurrence of these activities, because no global clock exists. As a result, it is very hard to replay the execution accurately. Fortunately, since the only kind of interaction between processes on different machines is by means of message passing, the ordering of the messages is sufficient to determine the execution pattern of the program. Recording of the fine grain interleaving of these processes down to the statement or instruction level is not necessary, because these activities cannot affect other processes anyway. Some algorithms [Lamp78] have been developed to determine the ordering of messages, but considerable run-time support is required.

Another common approach to debugging distributed program is the ***event-based*** approach. It follows the behavioral approach in keeping track of the execution history of the program. Events are any interesting activities of the processes specified by the programmer. Whenever such activities are detected by the debugger, one or more events will be activated. These events might carry information about the current status of the corresponding process. Analyzing the sequence of events and the order of their occurrences might lead to better understanding of the behavior of the program. Distributed programs are usually very large and generate an enormous amount of events. Therefore, the debugger should be able to filter out the irrelevant events and present the result in some comprehensible way to the programmers. The generation of an abnormal sequence of events might indicate an error in the program. Some kind of event recognition mechanism must be used to capture this error condition and report it to the programmers.

2.4. Related Work

A brief survey of some of the previous work on software debugging is given here. This is only a partial description of some of the significant work and is by no means exhaustive. More detailed information can be found in [ACM83, ACM88, ACM91, McDo89]. Only those debuggers that apply approaches related to our proposed design are listed. Other approaches to software debugging that are not discussed in detail are, for example, execution animation (either by time-process diagrams [Hart85], graphical representations of data structures [Shim91, Soch88], algorithmic animation [Brow88], or even sound [Fran91]) and static analysis [Alle86, Tayl80].

Most of the more popular debuggers for sequential programs provide more or less the same features, such as setting breakpoints, single stepping through instructions or statements, displaying source code and values of variables, and modifying the hardware registers or memory contents. Mark Linton's *dbx* [Adam86] provides facilities to trace the inferior process by printing out information whenever a certain statement is reached or the value of some user-specified variable is modified. However, variable tracing is implemented by stopping the inferior at every single instruction and then checking if any variables being traced have been modified. This method results in slowing down the inferior by several orders of magnitude, and is impractical in most cases. Richard Stallman's *gdb* [Stal91b] allows users to define a sequence of debugging commands to be executed automatically when the inferior process hits a certain breakpoint. It also allows declarations of debugger variables to record data retrieved from the inferior. These features could be viewed as a primitive form of programmable debugger language. Mark Russell's *ups* [Russ91a] provides a nice graphical interface that allows many debugging commands to be input by clicking the mouse buttons. In addition, *ups* incorporates a C language interpreter so that debugging statements written in C (but not functions or definition of global data structures) can be attached to any breakpoint.

Very little work has been done on studying how debuggers should be implemented to make them portable among different machine architectures and languages. Mark Linton [Lint90, Pan91] has addressed some experience gained in porting the *dbx* debugger to various UNIX systems, and how he envisions *dbx*'s functionality can be generalized to support performance monitoring and related tools. But no concrete mechanism or model is given on how this generalization can be done. Others [Fyfe91, Li91, Lopr89] have implemented toolkits or programming environments that provide primitives for building portable debugging or monitoring tools (see Section 6.4).

In terms of operating system support for debugging, Killian [Kill84] proposes an enhanced mechanism for interfacing the debugger and the inferior processes in UNIX systems. He suggests to use a special file system where each *file* represents an executing process. Examination and modification of the inferior processes' address space can then be made through the standard *seek*, *read*, and *write* mechanisms. This allows a much more efficient interaction between the debugging and the inferior processes. Mach [Acce86] is a newly designed operating system that has much stronger debugging support than existing systems. The Mach kernel provides a generalized exception handling mechanism [Blac88] for the debugger to detect any anomalies in an inferior process. Also, it provides a set of low-level primitives for shared memory and multi-thread process control that allows debuggers to access the inferiors' address spaces efficiently [Casw90].

Mellor-Crummey and LeBlanc [Mell89] propose a software solution for machines that do not support hardware instruction counter. They describe the implementation of a software instruction counter (SIC) that achieves the same effect of a hardware instruction counter by keeping a *backward branch count* and the program counter. They show that these two counters together is sufficient to uniquely describe the state of a process. Thus, the SIC can be used to guide execution replay or reverse execution.

McDowell and Helmbold have written an excellent survey on debugging concurrent programs [McDo89]. In this section, we only discuss some of the more important and more recent work. The majority of the work on parallel debugging applies the recording and replay approach. However, researchers have developed many very different methods to ensure reproducible executions. Garcia and Berman [Garc85] model the interactions between the processes of a parallel program by Petri nets, and use them as a form of high-level descriptions of the execution of the program when it is replayed. Recording of the execution is achieved by using special library routines that act as *watchdog* and issue software traps to the operating system whenever some pre-defined events (e.g., semaphore operations and process creations) occur. During the replay phase, the recorded information is used to produce a *movie* of the program execution. This is actually a series of Petri nets representing the process interactions being displayed in sequence at a speed controlled by the user. The disadvantage of this approach is that its execution replay is a passive, non-interactive activity. Only that information that has been recorded is available to the programmers. Furthermore, programmers have no control on the choice of events to be recorded, and the debugger only supports recording of events at the process interaction level.

Carver and Tai propose a solution to reproduce the execution of concurrent programs that consist of processes interacting only through semaphores and monitors [Carv86]. They model the execution of a parallel program by a sequence of P operations (*P-sequence*) on shared semaphores. A total ordering of these *P-sequences* suffices to characterize the interleaving between the processes of a parallel program. By forcing all semaphore operations to follow the same order as in the recorded *P-sequences* when the program is re-executed, the relative ordering of the processes will be the same as in the first run. Carver and Tai apply the same idea to programs that use monitors as synchronization primitives, using instead a *M-sequence* of calls to monitor procedures. However, this scheme requires the synchronization operations to be serialized. This unnecessarily strong constraint will affect the performance by limiting the possible parallelism of the program. Furthermore, when this serialization is removed in the production version, the behavior of the program might be different. In the distributed environment—where the total ordering of concurrent processes is almost impossible to obtain—this method requires modifications that might introduce extra overhead.

LeBlanc and Mellor-Crummey describe a more general approach to achieve reproducible executions [LeBl87]. Their method is not dependent on any particular form of interprocess communication mechanism. Instead, it requires the processes to follow a specific protocol whenever they access any shared objects. This protocol implements a concurrent-read-exclusive-write (CREW) access mechanism. Also, a version number is maintained for each shared object and it is incremented whenever the object is modified. Operations on shared objects and their version numbers are recorded. This can be viewed as a generalized version of Carver's approach: while the *P-sequence* defines a total ordering of access to shared semaphores, the pairs of read/write access and version number define a total ordering for write operations and a partial ordering for read operations on shared objects. The program is re-executed in a manner similar to that in Carver's approach, and the program's behavior can then reproduced. One disadvantage of both Carver's and LeBlanc's approaches is that they require the processes to call special routines (special P and V procedures and the CREW protocols) before accessing shared objects. The existence of these routines might hide away subtle bugs that will show up only when the routines are removed. In that case, other debugging methods have to be used.

In order to minimize the intrusiveness of the debugger, Aral and Gertner propose to take advantage of the multiprocessor hardware and let the debugger run concurrently with the inferior processes on a separate processor [Aral88b]. The use of shared memory allows the debugger to have full access to the complete core image of the inferior. Since the debugger is executing on a separate processor, the execution of the inferior is not affected (except possibly with a negligible delay due to memory contention). Through the use of dynamic linking, the debugger can access not only the inferior's memory content, but also directly execute functions defined in the inferior. By carefully modifying the machine instructions in the inferior process, the debugger can force the inferior to invoke debugging functions in the debugger's address space (see Section 5.2.2). This allows programmers to insert assertions or debugging statements without changing the source programs.

In the area of distributed debugging, many researchers believe that monitoring the passing of messages between the processes is all that is required to debug a program. Examining the internal states of each individual process would give programmers too much information that might not be necessary and might be hard to digest. Smith's debugger [Smit85] is one such example. Richard LeBlanc and Arnold Robbins, in their debugger *Radar* [LeBl85], support a monitoring facility that captures all message transmissions and process related activities. They also introduced an animation tool that can replay graphically the execution of the program based on the recorded information. A programmer can control the speed of the replay and examine the content of the messages, but has no access to the internal states of each processes.

By means of special system libraries, Elshoff's debugger for the Amoeba system [Elsh88] records the activities of the program in a way somewhat similar to the scheme used by [LeBl87] in parallel debugging. All system calls are trapped and recorded as *events* during the monitoring phase. When the program is re-executed, the debugger forces these calls to follow the recorded order. Since system calls are the only channels through which a process can access resources, such as secondary storage and inter-process communication mechanism, other than its own address space, a recording of all system calls is sufficient to reproduce the interleaving of all processes. In order to limit the number of events presented to the programmer, a finite state machine (FSM) model is used to capture only those event sequences that are recognized by this FSM.

Wittie [Witt88] approaches the repeatable execution problem by frequently checkpointing the executing processes. Periodically all processes of the program are halted and the states of each processes are captured. A special checkpoint algorithm is introduced so that messages being transmitted on the network while the program is halted will be delayed for the same length of time taken by the checkpoint operation. In this way, it creates the illusion that the checkpoint operation takes zero time. Program execution can then be rolled back to any previously recorded states. However, there is no mechanism to ensure that the relative ordering of the processes is preserved. Instead, records of the states are time-stamped and the accuracy of the replay is measured by the timing difference when compared with the records.

Bates and Wileden [Bate83] take the approach of viewing the *Behavioral Abstraction* of the inferior processes. The program is viewed as characterized by its activities rather than by its states. The activities of a program are represented by a stream of interesting *events*, which are defined by the programmer. Different levels of abstraction can be built up by combining selected primitive events to form higher-level events. Formation of high-level events is carried out by a pattern matching based event recognizer. When a specified sequence of primitive events is recognized, the corresponding high-level event is formed. The relationship between the high- and low-level events is described by the *Event Definition Language* [Bate82]. Baiardi *et al* [Baia86] describe a debugger for the language ECSP using a similar approach. They use a *Behavior Specification* language to describe the partial order on the events. However, their choice of primitives for specifying the program behavior is somewhat arbitrary. Bruegge and Hibbard [Brue83] describe the behavior of a program by means of *Path Expressions*, which are based on regular expressions and can be used to describe the flow of control of a program. Hseush and Kaiser [Hseu88] extend path expressions by adding the capability to describe the flow of data in the program. They are currently working on an implementation of this approach.

Chapter 3

The Dataflow Approach

In this chapter, we describe a new, dataflow approach to debugging. This approach provides a solution to the problem of collection and presentation of information about the inferior's execution to programmers, and ways for programmers to control a debugger easily and efficiently. To experiment with this approach, we have implemented a prototypical debugger called Dalek for sequential programs.

The foundation of our approach is event-based debugging. As described in Chapter 2, event-based debugging is a common method used in debugging concurrent programs [Baia86, Bate88, Elsh88, LeBl85]. These existing event-based debuggers usually provide a filtering mechanism to help eliminate irrelevant events. However, these mechanisms remain as passive filters and cannot generate feedback to the filtering mechanism or modify the filtering operation in response to the information collected. In other words, information that is not recorded directly, or cannot be derived from the recorded events, is unavailable. Furthermore, most of these debuggers only look at the interactions between processes and do not provide information about the internal state (such as content of variables, stack, frames, etc.) of individual processes. Thus, they cannot be used for debugging sequential programs or sequential parts of distributed programs. In addition, many debuggers that employ passive-filtering mechanism require special expressions [Bate88, Brue83] to describe the data to be filtered or retained. These expressions are sometimes complicated and confusing, and can only recognize those sequences of events that exactly match the given expressions. As a consequence, erroneous event sequences are hard to detect because their patterns are often not known in advance.

Our approach to event-based debugging employs user-defined dataflow graphs to describe how events are processed and correlated with each other. Nodes in a dataflow graph are active processing units that are activated by the occurrences of events. Also, appropriate combination of lower-level events might trigger higher-level events, which represent a higher-level abstraction of the behavior of the inferior process. This approach allows *active* event recognition. That is, the debugger can carry out arbitrarily complex actions throughout the steps in the event recognition. For example, current events can be correlated with old events to infer more meaningful information, or if necessary, the debugger can access the inferior process's address space to acquire further information. Based on this information, analysis can be carried out to decide whether a particular event should be recognized and propagated to higher levels. As a result, this dataflow approach is more flexible and powerful than the passive filtering schemes; it allows both normal and erroneous event sequences to be captured.

Furthermore, the actions carried out in each computational unit of the dataflow graph is defined by familiar programming language constructs. These programming language constructs are also uniformly used to control other operations of the debugger. In other words, programmers can specify any arbitrarily complex debugging operations that can be expressed by the given language.

The prototype debugger, Dalek, that we have implemented is based on *gdb* [Stal91b]. *Gdb* is a debugger that runs under the UNIX operating system and can be used for debugging C programs. Dalek's overall operation is similar to *gdb*'s. Dalek and the inferior run as separate processes, with Dalek controlling, observing, and altering the inferior's execution according to code written in the Dalek language. Borrowing a fair amount of code from an existing debugger and building Dalek on top of it allowed us to quickly build an experimental prototype without spending too much time on implementation details. On the other hand, Dalek inherits the limitations of *gdb* and thus runs only under UNIX and can be used only for debugging C programs. However, the dataflow approach is not bound to these limits and is independent of the operating systems and programming languages.

In the following sections, we describe the dataflow approach and Dalek's programming language. We also give examples of Dalek's code, and describe how the dataflow approach can be extended to distributed environments. Additional details can be found in [Craw90, Craw91, Olss90, Olss91a, Olss91b].

3.1. Event Recognition by Dataflow Methods

An *event* is an abstraction of any interesting activity that occurs in the inferior process. Each occurrence of such activity generates a data *token*, which is a representation of an *instance* of an event. Any number of attributes may be associated with each token. These attributes can be used to hold information related to their corresponding event instance. ***Primitive*** events are those triggered directly by activities within the inferior process (e.g., execution of a statement, or invocation of a procedure). ***High-level*** events are those triggered by the recognition of other events. Each high-level event contains a list of constituents, which are primitive or other high-level events on which it depends. When one or more of these lower-level events are recognized, the corresponding high-level event will be triggered.

Primitive events can be thought of as characterizing interesting changes in the control and data state of the inferior process. Typical primitive events include entry to or exit from procedures or smaller blocks of code, as well as changes in a program's data ranging from a change in a variable's value to more complex changes such as a linked list becoming empty, an array becoming unordered, or a graph becoming cyclic. Such complex changes, though, are most often expressed as high-level events.

The definition of a high-level event specifies which constituent events will trigger that high-level event. That is, the definition indicates which lower-level events' tokens, after arriving on incoming arcs, will activate the event's command block so it can attempt to recognize an instance of that high-level event.

When a high-level event is triggered, its associated code decides how it should react based on what constituents' tokens are present and the values of their attributes. The code might decide that the present constituents (or some subset thereof) comprise a valid instance of the high-level event. In such a case, to characterize this particular occurrence of the high-level event, the code should assign values to its own attributes—either by copying those of selected constituents or by computing some function based on the available data. Other possible actions include printing a message, reestablishing interactive control while keeping the inferior process suspended, and even modifying data or control flow in the inferior process. The code will then generally remove the selected tokens from its incoming queues so as to avoid repeated recognition of the same tokens when it is subsequently triggered. When the event's code finishes execution, tokens embodying the new event propagate to all high-level events that depend on it, and the recognition phase begins anew. On the other hand, the code might decide that the appropriate combination of constituents is not present on the incoming queues or that their attributes fail to satisfy the desired relationships.

In most models of dataflow, tokens flow between the nodes of a directed graph. In Dalek, the leaves of this graph represent primitive events (independent sources of tokens) and the interior nodes represent higher-level events. Initially, activity within the inferior process causes it to suspend (e.g., at a breakpoint), transferring control to the debugger. While the inferior program remains suspended, the debugger triggers the primitive event associated with the activity in the inferior; the debugger generally transfers control back to the inferior process when the computation in the dataflow graph reaches a quiescent state, i.e., when all triggered nodes have completed execution. If the primitive event is recognized, it generates a token. Since a token corresponds to an *instance* of an event, a token is a composite object; the information required to characterize a particular occurrence of an event is carried with it as its attributes. Recognition of a primitive event causes copies of a token characterizing that event occurrence to flow along dedicated arcs to all high-level events that depend on it.

In the dataflow model, computation is performed at the nodes. In Dalek, this computation is specified by the programmer using Dalek's programming language. Incoming arcs are represented as queues to which arriving tokens append themselves.

Figure 3-1 illustrates the dataflow approach to event recognition. A, B, and C represent activities (e.g., breakpoints) within the inferior process whose occurrence can trigger primitive events D, E, and F, respectively, as indicated by the dashed arrows. In turn, the occurrence of those primitive events can trigger higher-level events, as indicated by the solid arrows. For example, the occurrence of D can trigger event G. Note that the occurrence of E can trigger two events, G

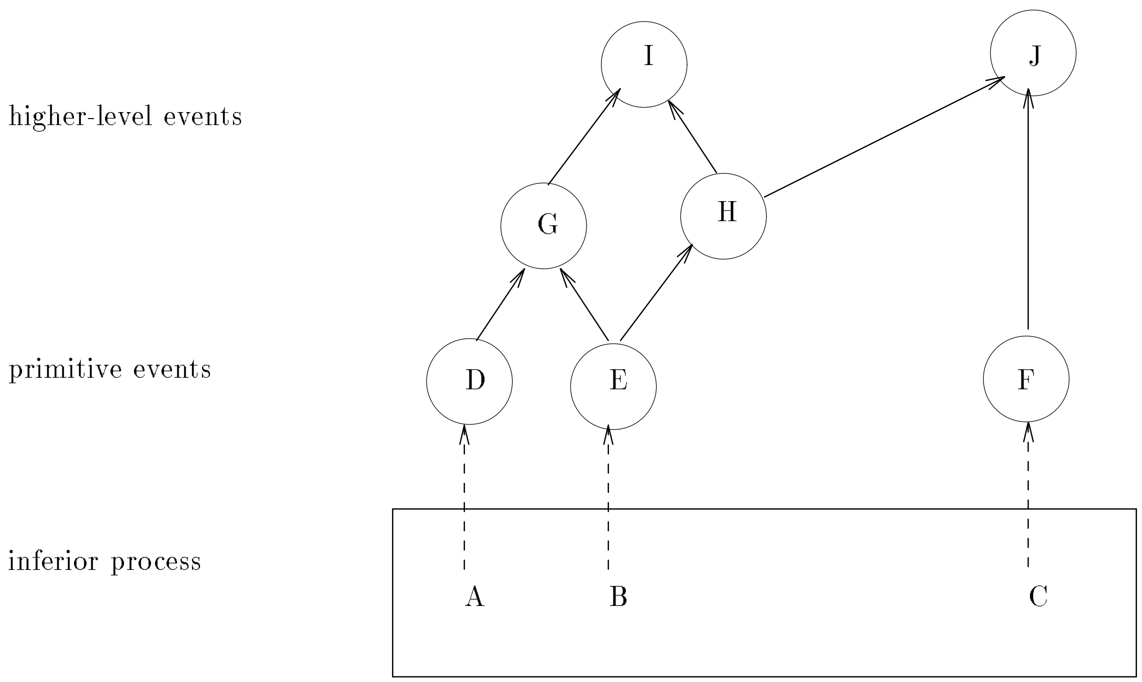

Figure 3-1. Event recognition by the dataflow approach.

and H. The two highest-level events, I and J, can be triggered by G and H, and F and H, respectively.

In order for an instance of a primitive event to occur, it must first be explicitly raised. Typically, the programmer defines a block of instructions and associates it with a breakpoint inserted in the inferior process. When execution of the inferior arrives at this breakpoint, its corresponding instructions are executed. These instructions should contain code that determines which primitive events should be raised, and collects the necessary data that constitute the event attributes.

3.2. The Dalek Language

Unlike most existing debuggers, Dalek includes a general-purpose programming language as part of its command language. This language is used to control the execution of the inferior process, specify events and their relationship, and define actions to be taken by each event node.

Language constructs provided by Dalek include assignment statements, a conditional statement, a looping construct, blocks with local variables, procedures and functions, as well as global variables. Dalek also provides ways to define, raise, recognize, and combine events. Status of the debugger and inferior process, such as the current value of the program counter or names of the functions in the call stack, can be accessed through built-in variables and pre-defined functions.

A programmable debugger is a very powerful tool. Quite often during debugging, a programmer might think of a few simple procedures that would be extremely helpful, e.g., a routine to traverse and print selected elements of a linked list. Injecting such debugging code into the source code, perhaps by conditional compilation, is not an attractive solution. Often, the programmer has arduously brought the inferior process to a critical point in its execution under the debugger's control before discovering the need for such a routine. Furthermore, seemingly minor modifications to the source code can substantially alter the symptoms exhibited by bugs, especially if pointers are involved.

Dalek gives programmers the ability to write debugging code when they find it necessary. When execution of the inferior process is suspended at a breakpoint, a programmer can direct control to be passed to such debugging code, either from the command level or automatically upon hitting the breakpoint. The Dalek code and variables reside within the debugger's address space. Thus, storage of these objects does not affect and is not affected by the inferior process. Furthermore, Dalek allows programmers to effectively replace code in the inferior process with code written in Dalek. A programmer may specify a breakpoint right before the code to be replaced, and associate Dalek code with that breakpoint. When the inferior hits the breakpoint, the associated Dalek code is executed. The last action of this Dalek code should direct control to the point right after the replaced code in the inferior process. This alternative to patching machine code has a temporary effect, affecting only how the code being debugged executes while under control of the debugger; it does not modify the program's executable file or the source code. Such temporary code patching can be used to fix multiple errors without exiting the debugger and recompiling the source code.

In the following sections, we illustrate the Dalek language and its applications by a series of examples.

```
# Dalek procedure to print out value field of nodes in a
# circularly linked list (in the inferior) starting at $head.
# Assumes $head is a dummy node, whose next field points to
# itself if the list is empty.

function $print_list($head)
    {
    local $p
    set $p = $head->next
    while ($p != $head)
        printf "%d\n", $p->value
        set $p = $p->next
    endwhile
    }
end

# Code invoking the $print_list function
break goo.c:27
commands
    silent
    call $print_list (symtab_head)
    cont
end
```

Figure 3-2. A Dalek procedure to print a linked list and its invocation.

3.2.1. Breakpoints and Functions

To illustrate the Dalek language, consider the Dalek code in Figure 3-2. The code defines a `$print_list` procedure, which can be invoked interactively from Dalek's command level or from any point during code execution by simply setting a breakpoint whose commands include a call to this procedure. In this particular example, `$print_list` is invoked from a breakpoint at line 27 of the file `goo.c`. The `silent` command suppresses the default announcements of encountering a breakpoint, the `call` statement calls the `$print_list` procedure, passing symtab_head as an argument, and cont continues execution of the inferior process. Dalek's variable and function names are prefixed with the dollar sign, in order to distinguish them from those names defined by the inferior. This example also illustrates a local block, enclosed within '{' and '}', that declares a local debugger variable $p.

We could have written the body of $print_list in-line as commands, but we wrote it as a function so it can be invoked directly from the command level or from another breakpoint, if desired.

Arguments and local variables of functions are stacked. Therefore a Dalek function can be called recursively. Figure 3-3 shows a simple recursive function, $ptree, that outputs in-order the nodes of a binary tree.

```
# Dalek code to print out a binary tree; its argument is a
# pointer to a subtree.
function $ptree ($p)
    if ($p != 0)
       call $ptree ($p->left)
       printf "%c\n", $p->item
       call $ptree ($p->right)
    endif
end
```

Figure 3-3. Recursive tree traversal in Dalek.

3.2.2. Automatic Program Tracing

Similar to the `$print_list` example, a programmer can write Dalek code to ensure that the inferior maintains certain invariants about its data structures. For example, code to check that a circularly linked list is indeed circular could be invoked on entry to and exit from each procedure that manipulates the list, or even at every step of the inferior. This approach is more flexible than using compiled-in assertion checks (e.g., like the *assert* macro in C) since what invariants to check and where to check them can be decided as the programmer finds it necessary during actual debugging. The invariant checking code can be saved for use in subsequent debugging sessions.

To demonstrate further how Dalek's mechanisms can be used to attack a debugging problem, consider a program containing a module that maintains a sorted linked list. This sorted list is kept local to the module and it cannot be accessed directly anywhere out of the scope of this module. All operations on the list are performed through a few explicitly exported functions. When this module is combined with other parts of the program and the program executed, somehow the contents of this list are corrupted. Careful examination of the list module leads us to believe that it is not the source of the corruption. Therefore, some other part of the program must contain erroneous code that has overwritten this linked list. This kind of bug is quite common in programs written in languages such as C that allow the use of pointers. Most likely it is caused by a pointer incorrectly referencing a memory location occupied by the linked list.

The debugging problem is to locate the erroneous code since it could be in any part of the program. To narrow down the possible location of the bug, we will silently trace, step by step, the program's execution.

```
# Function to verify if a linked list is sorted in
# non-decreasing order (i.e., by <=).  It returns 1 if the
# list is sorted, 0 otherwise.

function $list_sorted ($head)
    {
    local $p, $previous_value
    set $p = $head->next
    if ($p != 0)
       set $previous_value = $p->value
       set $p = $p->next
    endif

    while ($p != 0)
       if ($previous_value > $p->value)
           func-return 0
       endif
       set $previous_value = $p->value
       set $p = $p->next
    endwhile

    func-return 1
    }
end

break initialize_sorted_list
commands
    silent
    set $head_ptr = list_head
    while ($list_sorted ($head_ptr))
       step
    endwhile
    printf "list has been corrupted\n"
    backtrace
end
```

Figure 3-4. Dalek code to uncover a list-corruption bug.

Figure 3-4 shows a piece of Dalek code that automates the program tracing. The function `$list_sorted` verifies the validity of the sorted list and returns zero when the list is no longer sorted. A breakpoint is set in the function `initialize_sorted_list`, the place where the sorted list is first used. The command block associated with this breakpoint begins with a loop that single-steps execution of the program, stopping when the list is first found to be corrupted. It then prints a trace of the current call stack. Note that in this command block, the value of `list_head` is saved in Dalek's variable `$head_ptr`, because when the execution steps into different scopes, the name `list_head` might no longer be accessible. Also, we consider this kind of debugging to be the "last resort", i.e., we use this technique after failing with the more standard ones.

The key to success here is that Dalek allows single-stepping within `while` loops. Hardware *data watchpoints*, available in some architectures, could also be used to help solve this kind of debugging problem. They allow specified memory cells to be monitored, generating an interrupt when such a cell is read or written. However, some means of filtering through the enormous number of generated interrupts would still be needed.

3.2.3. Primitive Events

Figure 3-5 presents a simple example of how a primitive event is defined and raised. The event `‘element` is defined to have three attributes, `‘v`, `‘nextv`, and `‘kind`; its command block is not shown. Note that event names and attributes are prefixed with a backquote to distinguish them from debugger variables and inferior process variables. Figure 3-5 also shows a breakpoint set at line 27 of the source file `foo.c`. Dalek suspends execution of the inferior process when it reaches that line; it then executes the command block. The command block raises the primitive event `‘element` by the command `event raise` in one of two ways depending on the values of variables in the inferior process.

```
# Definition of primitive event ‘element; its attributes are
# ‘v, ‘nextv, and ‘kind.
event define ‘element (‘v, ‘nextv, ‘kind)
    # The command block for primitive event ‘element
    . . .
end

break foo.c:27
commands
    silent

    # Raise the ‘element event with one of two sets of attributes
    if (nodeptr->next)
       event raise ‘element (nodeptr->value, nodeptr->next->value, "middle")
    else
       event raise ‘element (nodeptr->value, "none", "tail")
    endif
    cont
end
```

Figure 3-5. Outline of defining and raising a primitive event.

3.2.4. Stack

This example demonstrates how high-level events are specified. It shows Dalek code that is used to monitor the operation of a stack with the usual push and pop operations. In the inferior process, the stack is implemented by an array called `stack` with the variable `stack_pointer` recording the offset of the top of stack from the beginning of the array. Figure 3-6 shows the definition of primitive events `‘push` and `‘pop`, and of the higher-level event `‘push_pop`. When either `‘push` or `‘pop` is activated, the current value of `stack_pointer` is recorded in its attribute `‘stack_p` (using the `event set-attribute` command), which is then passed to `‘push_pop`. In addition to its own name and no attributes, the defini-

```
# This event is raised when push entered; it records
# stack_pointer as its attribute.
event define ‘push (‘stack_p)
    event set-attribute ‘stack_p stack_pointer
end

# This event is raised when pop entered; like ‘push, it
# records stack_pointer as its attribute.
event define ‘pop (‘stack_p)
    event set-attribute ‘stack_p stack_pointer
end

# Correlate a ‘push with a ‘pop.  This event is triggered only
# by arrivals of ‘pop.  Its queue for constituent event ‘push
# stacks yet-unmatched ‘push’s.  When ‘push_pop executes, its
# queue for constituent event ‘pop contains only the most
# recent ‘pop.
event define ‘push_pop () –‘push ‘pop
    # Ensure stack pointer was same for push and corresponding pop
    if ($value(‘push, 0, ‘stack_p) != $value(‘pop, 1, ‘stack_p))
       printf "Stack pointer error: pushed at %d, popped at %d\n", \
              $value(‘push, 0, ‘stack_p), $value(‘pop, 1, ‘stack_p)
    else
       printf "Pushed and popped %d, stack_pointer = %d\n", \
              stack[stack_pointer], $value(‘push, 0, ‘stack_p)
    endif

    event remove ‘push, 0   # Discard instances of ‘push
    event remove ‘pop, 1    # and ‘pop from this event’s queue.
end
```

Figure 3-6. Events to monitor stack activity.

tion of ‘`push_pop` names two constituents, ‘`push` and ‘`pop`, the lower-level events on which it depends. The arrival of a token from a constituent event whose name is prefixed by ‘–’ will not cause the commands for the high-level event to be executed, although the token will still be queued on arrival. Hence, only the arrival of a token from ‘`pop` triggers execution of ‘`push_pop`; the arrival of a token from `‘push` does not.

The code associated with `‘push_pop` reflects the last-in, first-out stack discipline. It maintains the following invariant on its token queues: Whenever it is not active, its `‘pop` queue is empty and its `‘push` queue contains, in arrival order, all `‘push`'s that have not yet been matched with a `‘pop`. The code first checks that the `‘stack_p` attributes passed on from the most recent `‘push` and `‘pop` are the same. These attributes reflect the values of `stack_pointer` when the item was originally pushed and later popped. Should their values differ, `‘push_pop` prints an error message. Otherwise, it prints the recorded `stack_pointer` value and the contents of the top of stack. The code uses the pre-defined function `$value` to access its constituents' tokens' attributes.

```
# Raise ‘push when push entered
break push
commands
    silent
    event raise ‘push
    cont
end

# Raise ‘pop when pop entered
break pop
commands
    silent
    event raise ‘pop
    cont
end
```

Figure 3-7. Activating the primitive stack events.

`$value`'s three arguments specify the name of the constituent, the position in the high-level event's queue for that constituent, and the attribute name. (Queue positions are numbered starting with 1 denoting the oldest arrival; as a convenient special case, 0 denotes the most recent arrival.) After checking the `'stack_p` attributes, `'push_pop` removes the corresponding constituent events' tokens from its queues using the `event remove` command, which uses the same position numbering as `$value`.

As seen in Figure 3-7, the primitive events `'push` and `'pop` are activated by an `event raise` command from within breakpoints set inside the `push` and `pop` functions respectively. After the `event raise` command is executed and the dataflow graph once again becomes quiescent, the `cont` command instructs the inferior process to continue executing. Figure 3-8 shows the dataflow graph corresponding to the code presented in Figures 3-6 and 3-7.

3.2.5. Memory Allocation

This example uses events to monitor allocations and deallocations of memory. Misuse of memory allocation and deallocation routines (*malloc* and *free* in C terminology) is a common source of program error, e.g., passing the deallocation routine an invalid address, especially one for a block that was previously freed. This example shows how errors in the use of the memory allocator can be detected.

Figure 3-9 shows the definition of two primitive events, `'malloc` and `'free`, which correspond to invocations of the allocation and deallocation routines. The event `'malloc` records in its attribute `'address` the address of the memory block

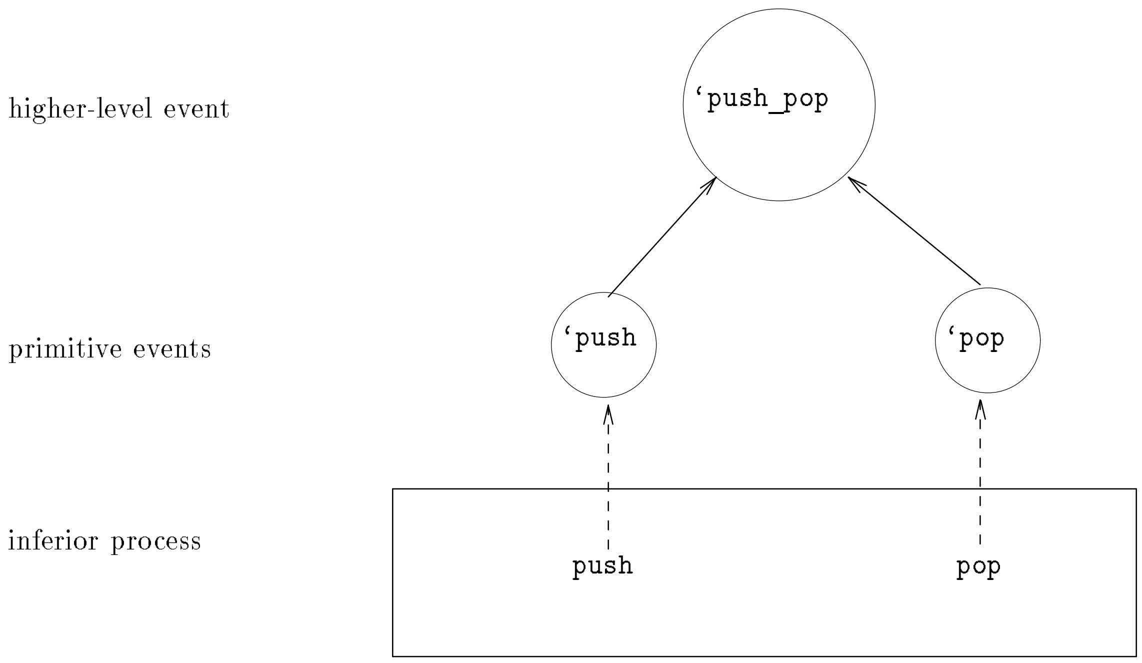

Figure 3-8. Dataflow graph for stack example.

allocated; `‘free` records in its attribute `‘address` the address of the memory block to be deallocated. The Dalek pre-defined function `$quiet_finish` resumes execution of the inferior process until it returns from the current procedure. This function then returns the procedure's return value. The debugger global variable `$addr` is set inside *free* to the address of the block being freed (i.e., *free*'s argument). Since *free* is a system-dependent library routine, how `$addr` is set is also system-dependent.

Error detection is carried out in the high-level event `‘match`. This event keeps track of the addresses of all memory blocks that have been allocated but

```
# The ‘malloc event is recognized each time it is
# triggered/raised by default; the same applies to the ‘free
# event.
event define ‘malloc (‘address)
    event set-attribute ‘address $quiet_finish()
end

event define ‘free (‘address)
    event set-attribute ‘address $addr
end


# The high-level event ‘match depends on lower level events
# ‘malloc and ‘free but it is triggered only by ‘free.
event define ‘match () —‘malloc  ‘free
   {
   local $qindex, $free_addr

   # $qlen returns the length of specified event queue
   set $qindex = $qlen(‘malloc)
   set $free_addr = $value(‘free, 1, ‘address)

   while ($qindex > 0)     # Search for ‘malloc matching this ‘free

       if ($value(‘malloc, $qindex, ‘address) == $free_addr)

           # Remove first token from ‘free’s queue
           # and $qindex’th token from ‘malloc’s
           event remove ‘free, 1
           event remove ‘malloc, $qindex
           wbreak   # Break out of the while loop
       endif

       set $qindex--
   endwhile

   if ($qindex == 0)
       printf "Error: attempt to free unallocated address 0x%x\n", $free_addr
       event remove ‘free, 1
       event dont-propagate
   endif
   }
end
```

Figure 3-9. Memory allocator monitor.

not yet matched with a deallocate request. Whenever `'free` occurs, `'match` checks to see if the corresponding `'malloc` event has been raised. If a `'malloc` with such an address cannot be found, `'match` prints an error message. Note that the order of freeing blocks need not be the same as that in which they were allocated.

As shown, `'match` is triggered by `'free`. The matching of `'malloc` and `'free` events is performed using a simple `while` loop, which sequentially compares the attribute `'address` passed by `'free` with those returned by the previous `'malloc` events.

This example can be extended to determine whether the reason for a mismatched *free* is that the block is being freed a second time. To do so, `'match` requires only small modifications. An attribute, say `'maddr`, needs to be specified in its definition and set when a match is found. Also, a loop needs to be added to the body of the second `if` statement. The loop simply searches the event history list for a `'match` whose `'maddr` attribute is equal to `$free_addr`. With additional attributes in `'malloc` and `'free` recording the names of the inferior process's procedures that invoke *malloc* and *free*, the crucial transition from merely detecting the symptoms of a bug to discovering its cause can be made. Searching the history list is also useful in different contexts. Suppose, for example, that the user has detected that the contents of an allocated structure are corrupted. Then, a programmatic search of the history list can determine whether the address of that structure was freed.

```
#include "lib/csim.h"            /* defines CSIM types, procs, vars: MBOX,
                                    EVENT, event, create, clock, etc. */
#define STM 2.0                  /* mean send time */
#define NUM 50                   /* number of items */
#define SENDERS 5                /* number of senders */

MBOX mb;                         /* mail box for messages */
EVENT go_ahead;                  /* tell sender to start */

sim()
{
    int i, msg;
    mb = mailbox("mail box");
    create ("master");           /* make this a CSIM process */
    go_ahead = event ("Start sending message");
    for (i=1; i<=SENDERS; i++)
       sender();                 /* initiate processes */
    hold (1.0);                  /* wait for creation of all senders */
    set (go_ahead);              /* signal all senders to begin */
    for (i=1; i<=NUM; i++) {
       receive (mb, &msg);       /** NOTE: Dalek event `recv is raised here **/
       printf ("Received %d from %d\n", msg % 100, msg / 100);
    }
} /* sim */

sender ()
{
    int i, me;
    create ("sender");           /* make this a CSIM process */
    me = identity();
    wait (go_ahead);
    for (i=1; i<=NUM/SENDERS; i++) {
       printf ("Sent %d from %d\n", i, me);
       hold (expntl(STM));       /* hold message transmission time */
       send (mb, me*100+i);      /** NOTE: Dalek event `send is raised here **/
    }
} /* sender */
```

Figure 3-10. A simple simulation program in C.

3.2.6. Pseudo-Concurrent Program

This example shows the usefulness of Dalek in debugging a pseudo-concurrent simulation program. Pseudo-concurrent programs, though deterministic, are similar to true concurrent programs. They consist of multiple simulated processes executing in parallel and sharing global variables. Furthermore, the exact interleaving of process execution is hard to predict. Such programs are obviously difficult to debug.

The simple simulation program shown in Figure 3-10 uses the CSIM simulation package [CSIM87]. It consists of six processes: one is the "master", each of the others is a "sender". Each sender repeatedly sends a message back to the master; the delivery of each message takes a random period of time.

One difficulty in debugging such a program is the large number of interactions between processes. It is difficult to isolate the relevant interactions. A common practice is to insert into the source program debugging statements to print out information about the current state of the process. However, these debugging messages will be intermixed in pseudo-concurrent programs. Filtering out irrelevant messages can be tedious. Using Dalek, however, such filtering is easily accomplished. Figure 3-11 shows two primitive events, `‘send` and `‘recv`, and the higher-level event `‘message`. The event `‘send` is raised when a message is sent by sender 3; `‘recv` is raised when the master receives a message from sender 3. (For simplicity, the sender number is encoded in the message.) Both events `‘send` and `‘recv` discard all other messages. When `‘message` matches a pair of `‘send` and `‘recv`, it outputs information regarding the match. If `‘message` detects a difference in the contents of the message sent from that received, it prints an error message. As indicated in Figure 3-10, `‘send` and `‘recv` are raised from breakpoints associated with the source code.

```
# Events ‘send and ‘recv are raised from breakpoints as
# indicated in Figure 3-10; each propagates
# (recognizes) only messages from sender number 3.

event define ‘send (‘from, ‘msg)
    if (me == 3)
        event set-attribute ‘from me
        event set-attribute ‘msg i
    else
        event dont-propagate
    endif
end

event define ‘recv (‘from, ‘msg)
    if (msg / 100 == 3)
        event set-attribute ‘from msg/100
        event set-attribute ‘msg msg%100
    else
        event dont-propagate
    endif
end

# Correlate ‘send and ‘recv by checking message contents

event define ‘message () —‘send ‘recv
    if (($value(‘send, 1, ‘from) == $value(‘recv, 1, ‘from)) && \
        ($value(‘send, 1, ‘msg) == $value(‘recv, 1, ‘msg)))
       printf "Message %d from %d completed. Time=%lg\n",\
              $value(‘send,1,‘msg), $value(‘send,1,‘from), clock
    else
       printf "Error:: sender: from=%d, msg=%d,", \
              $value(‘send, 1, ‘from), $value(‘send, 1, ‘msg)
       printf "receiver: from=%d, msg=%d\n", \
              $value(‘recv, 1, ‘from), $value(‘recv, 1, ‘msg)
    endif

    event remove ‘send, 1   # Remove first token from queue for ‘send
    event remove ‘recv, 1   # and first token from queue for ‘recv
end
```

Figure 3-11. Dalek code to filter and match sends and receives.

3.3. Other Applications

Dalek's debugging language makes it suitable for many applications, three of which are presented here: procedure tracing, performance measurement and code profiling, and audit trails analysis. These applications further show the versatility and power provided by Dalek's full programmability and its dataflow mechanisms.

3.3.1. Procedure Tracing

Dalek does not provide a separate "trace" command; instead, it provides a more general mechanism—*broadcast breakpoints*—that, in conjunction with its full programmability, makes obtaining a trace of selected procedure calls very simple as a special case. Broadcasting lets programmers give a single command to establish an ordinary breakpoint at the entry to every procedure whose name matches a given regular expression, and associate a common block of debugging commands with those breakpoints. Without a broadcast breakpoint facility, programmers would have to, for each procedure, establish a breakpoint and give the associated breakpoint code—a rather tedious activity.

Thus, to have Dalek print out the names of all procedures entered during program execution, a programmer can just specify a single broadcast breakpoint command with '*' as its regular expression and the following code as its command block:

```
silent
printf "%s\n", $func(0)
cont
```

The pre-defined Dalek function `$func(N)` returns the name of the inferior function associated with the `N`-th frame of the procedure call stack. When `N` is `0`, as

```
# Broadcast breakpoints.  The commands compute conditional
# probabilities.  Initialize some Dalek Variables.  These
# variables are global in scope.

set $last="none"
set $push_push = $push_pop = $pop_pop = $pop_push = 0
set $running = 1

# broadcast breakpoints
proc-break *
    silent
    # Execute for every procedure entry
    if ($last == "push")
       set $push_push++
    else if ($last == "pop")
       set $pop_push++
    endif

    set $last = "push"

    if ("_exit" == $func(0))  # end of execution.
       set $running = 0
    else if (0 == $top_frame())  # already on outermost frame.
       cont
    else
       call $quiet_finish()
       if ($last == "pop")
           set $pop_pop++
       else if ($last == "push")
           set $push_pop++
       endif

       set $last = "pop"
    endif
end

# Start execution of the program:
run
while ($running)
    cont
endwhile

# Print out one result:
printf "Conditional prob. that PUSH is followed by a PUSH = %f\n," \
       (float) $push_push / ($push_pop + $push_push)
```

Figure 3-12. Computing conditional probabilities for stack usage.

in this case, it returns the name of the procedure in which the inferior is currently executing.

More selective tracing is also possible by specifying a more restrictive regular expression or by refining the code in the broadcast breakpoint's command block. Two examples of the latter include

- printing only when a certain procedure appears somewhere on the call stack (the call stack can be searched using a `while` loop)
- printing only when the value of some debugger variable satisfies an arbitrary Boolean expression (the value of this variable might be conditionally modified by the code associated with events or other breakpoints)

We have used broadcast breakpoints to obtain information used in correlating the activities by which programs access their run-time (procedure) stacks. This information is important in assessing the utility of certain architectural features. The particular conditional probabilities we needed to measure were: a push of a stack frame followed by the push of another stack frame, a push followed by a pop, a pop followed by a push, and a pop followed by another pop. Figure 3-12 illustrates how we programmed Dalek to gather and compute such information. We have also used broadcast breakpoints to solve debugging problems like the one presented in Section 3.2.2. Broadcast breakpoints are faster than the single stepping employed there; however, their granularity is coarse since they occur only when a procedure is entered. Therefore, using them will give only the name of the routine invoked *after* the routine in which the contents of the list were corrupted, rather than the source statement actually causing the corruption. Thus, the fastest way to zero in on such a bug is for the broadcast breakpoint commands to increment a counter to determine the number of times procedures are invoked, rerun the inferior until one procedure call before that state, and only then initiate the `while` loop that single steps.

3.3.2. Performance Measurement and Code Profiling

Dalek can be used to time parts of a program and to profile sections of code. A considerable advantage in using Dalek for these roles is that the part of the program for which information is to be gathered, as well as the precise information to be gathered, can be specified interactively and dynamically, without modifying, recompiling, and relinking the inferior's source code.

Dalek provides three pre-defined functions that return the cpu time used by the inferior process, corresponding to how UNIX reports cpu times. *User* time is the time the cpu spends executing the instructions in the inferior process's address space, whereas *system* time is the time the cpu spends executing system calls invoked by the inferior; *total* time is the sum of user time and system time. The function `$utime` returns the inferior's user time, `$stime` returns the inferior's system time, and `$ttime` returns the inferior's total time. Unfortunately, only `$utime` is accurate; `$stime`, and therefore `$ttime`, is not accurate because some of the cost of the debugger controlling the inferior is charged as system time to the inferior process. Furthermore, even `$utime` is not always accurate on all machines. For example, on machines that employ instruction pipelining, `$utime` can be affected by what instructions are in the pipeline; the pipeline is flushed when the operating system switches contexts, which occurs in alternating execution between the inferior process and the debugger. However, these functions can still be used in the debugger code that is executed as part of a breakpoint or event, and therefore exactly what parts of the inferior are monitored can easily be changed interactively and dynamically.

One use of Dalek's timing functions is for tuning a program's performance. For example, suppose we want to determine the times for both successful searches and unsuccessful searches of a symbol table. This information can give us feedback on the appropriateness of our data structure representation, perhaps suggesting that a hash table would perform better than a linked list. To obtain these times, one breakpoint can be set on entry to the search function and another breakpoint can be set right before the search function returns. The first breakpoint records the starting user time; the second adds the difference between the current user time and that recorded starting user time to the running sum for either successful searches or unsuccessful searches, according to the value the search function is returning. Alternatively, the breakpoints could raise events that include the time information as attributes. That would allow a more detailed analysis, say to compute averages and variances, since each instance of an event is recorded on the event history.

Gathering timing information using Dalek, instead of using standard profiling tools—such as UNIX's *gprof* tool—or including timing code in the source program, provides the user with more control over what is being measured and is not intrusive. As illustrated in part by the above example, individual or groups of procedures or statements can be measured, using arbitrary conditions to select those of interest. Profilers do not provide such fine-grained control. Including timing code in the source program is viable, but that requires recompilation and can affect overall user time measurements. The number and kinds of changes to the source code required by that approach can also become unwieldy. For example, suppose we want to find the cumulative time spent executing in procedure `P` but only when it is invoked directly or indirectly from procedure `Q`. To do such measurements by modifying the source code would require either an additional parameter in procedure `P`'s interface or a global variable; either of these changes would be cumbersome in a program of any size. Dalek code to do such measurements employs the `$func` pre-defined function, described earlier, to determine if procedure `Q` is on the procedure call stack.

Dalek can also be used as a simple code profiler and to gather other useful statistics concerning program behavior. For example, Dalek has been used to count the number of times a given procedure or statement is executed, to find the maximum length a linked list reaches during execution, and to measure run-time stack frame behavior, as seen earlier in Figure 3-12. This kind of information is easily obtained by setting breakpoints at appropriate points in the code and specifying breakpoint commands to maintain debugger variables; control statements and functions are used for performing the more complicated measurements.

3.3.3. Audit Trails Analysis

Being a powerful tool for event recognition and analysis, Dalek can be used not only as a debugger, but also as a stand-alone data analyzer. Recall that when used as a debugger, primitive events are generated by interesting activities occurred in an inferior process. These primitive events are then fed into Dalek's dataflow mechanism for analysis. Similarly, primitive events obtained from other sources can also be directed to Dalek. In other words, we may ignore Dalek's ability to monitor an inferior process, and use only its dataflow mechanism as a general-purpose data analyzer.

Any source of data that can be modeled as a stream of primitive events can be analyzed by the dataflow mechanism. For example, different types of transactions in a banking system can be defined as primitive events, and each individual transaction as an instance of a primitive event. As another example, we may define each operating system service as a primitive event and the stream of system calls issued by a process can be fed into Dalek for analysis.

We have successfully used Dalek to analyze audit trails obtained by a network security monitor (NSM) [Hebe90]. NSM is a tool that monitors activities across a local area network. Typical activities reported by the NSM include connection (remote login) or disconnection (logout) between machines, reading or writing of remote files, or execution of commands on a remote machine. One of the objectives of analyzing the NSM output is to detect malicious activities such as break-in attempts or violation of security policies. Such activities can be characterized by a combination of a number of basic network operations. By defining the output of the NSM as a stream of primitive event instances, Dalek's dataflow approach can be used to detect those sequences of actions that might constitute a security threat.

Figure 3-13 shows an simple example of using Dalek's dataflow approach to detect two malicious activities, namely, attempts to login to a remote machine as "guest", and attempts to remote login from one machine to another via a third machine. Four types of output from the NSM are defined as primitive events. They are shown as leaf nodes in the dataflow graph. Node "*guest*" represents a request from a remote machine to login as guest. Nodes *telnet* and *rlogin* both denote a connection request from one machine to another, using the telnet and rlogin protocol respectively. Node "*rlogin*" represents that a rlogin command is

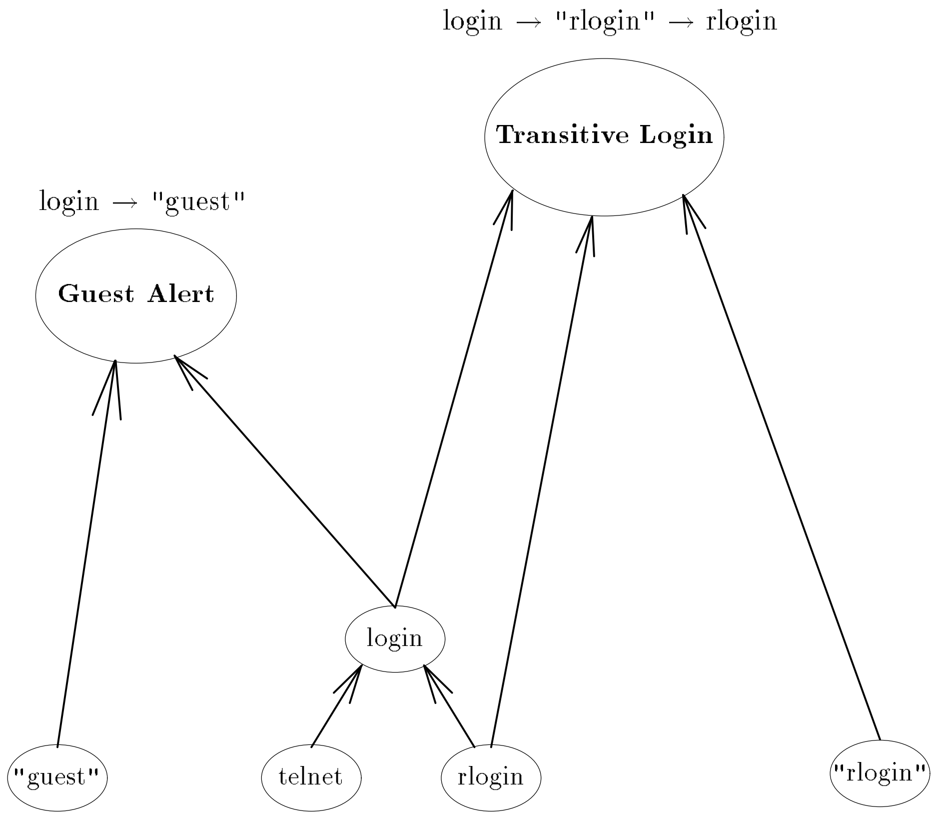

Figure 3-13. Detection of guest and transitive logins.

given from a certain machine.[1] Each of these primitive events carries as attributes a unique network session identification number, the name of the source machine sending out the request, and, in the case of *telnet* and *rlogin*, the name of the destination machine for the connection request.

[1]The current implementation of the NSM simply searches for specified keywords being sent across the network. Therefore, any string that contains words such as "guest" or "rlogin" will be reported as guest login request or remote login respectively.

```
event define `Guest_Alert () -`login +`guest
    {
       local $pos_login, $con

       set $con = $value(`guest, 0, `connection)
       set $pos_login = $qlen(`login)

       # search for the matching login event with the same connection id.
       while ($pos_login > 0)
           if ($value(`login, $pos_login, `connection) == $con)
              printf "Guest login alert in session %d from %s to %s.\n", \
                  $con, $value(`login, $pos_login, `from), \
                  $value(`login, $pos_login, `to)
              wbreak
           else
              set $pos_login--
           endif
       endwhile

       event remove `guest, 0
    }
end
```

Figure 3-14. Dalek code for recognizing a remote guest login.

Three high-level events are defined. Node *login* represents a remote login request regardless of the protocol used. It is recognized whenever either *telnet* or *rlogin* is triggered. Event *Guest Alert* is recognized whenever a login event is followed by "*guest*". Code in the node *Guest Alert* verifies the source and destination machines corresponding to each incoming event and matches them up accordingly, as shown in Figure 3-14. The definition of a *Transitive Login* from machine A to B and then to C is a login connection from machine A to machine B, followed by a rlogin command issued from machine B, and then followed by a login event from machine B to machine C.

More complicated malicious activities can be detected by specifying the appropriate dataflow graph and code in each node.

Other than the implementation shown in Figure 3-14, which uses pure Dalek code, we have also built another prototype that performs exactly the same function but is implemented entirely in the C programming language. In this prototype, the event filtering mechanism and relationship between events are "hard-coded" and cannot be changed without modifying and recompiling the source program. However, this prototype is almost an order of magnitude faster than the pure Dalek code version. Hard-coded implementations are usually useful for post-mortem analysis of the audit trials, but for interactive or on-line analysis where modifications of the event definitions and dependencies are necessary, the Dalek version should be used.

3.4. Drawbacks and Improvements

Dalek has several drawbacks that need to be improved. The kind of high-level debugging encouraged by Dalek requires a fair amount of interaction between the debugger process and the inferior process. This kind of interaction, unfortunately, is generally expensive since it requires system calls and their concomitant context switches. However, although such high-level debugging might require a few minutes of computer time, it can save hours of programmers' time. Nevertheless, in Chapter 5 we will describe ways to greatly reduce the overhead in such interaction.

Another potential drawback of *any* approach to debugging is that the debugger code intended for debugging the inferior may itself contain bugs. Dalek code is as susceptible to bugs as any other debugging code, perhaps more so since it supports and encourages the use of more complex features. How can instructions written in the debugger language be debugged? It might seem that undertaking two-stage debugging is a possible solution, but in reality such tandem-debugging only complicates matters even further. Early in Dalek's development, one copy of Dalek was used to help debug Dalek language code being interpreted by another copy of Dalek, which was in turn overseeing the execution of an application program. That experience indicated unequivocally that such two-stage debugging is not a viable solution. Instead, Dalek supports several "introspection" facilities so the user can interrogate it regarding its state.

One Dalek introspection facility is verbose mode, which can be selectively enabled by the user. In this mode, Dalek displays every command (and the name of its enclosing block) as it is interpreted. Making the debugger's actions visible in this way allows many simple errors to be found easily.

Dalek also provides several means to pause its execution during interpretation of the user's debugging program. The user can then examine or alter the state of the inferior process and, equally importantly, that of the debugging program itself. When the debugging program is paused inside an event, for example, values of that event's attributes can be manually inspected and altered. The debugging program's logic can be overridden by breaking out of an enclosing `while` loop or by forcing a user-defined debugging function to prematurely return a specified value. In addition, any block of code in the debugging program can be edited before resuming interpretation.

Dalek will pause in its interpretation of debugger commands when it encounters a `pause` command or when the user types the interrupt character, e.g., control-C. Using this latter method obviously does not afford the user much choice over the line of debugging code at which interpretation pauses. To pause at a particular line of debugging code, the user can set a meta breakpoint at any line in the Dalek language program.

For tighter control, "dstep" mode allows meta single-stepping. Prior to interpreting each line of debugging code, dstep mode displays the same information as is shown in verbose mode, but in addition it offers the user the option of pausing, editing the current command block, or skipping the displayed command entirely.

Another needed improvement is to raise the level of the Dalek language. The current language is intentionally low-level to ensure sufficient power and flexibility for experimentation. However, its low level sometimes makes it cumbersome to use, e.g., the way that the event queue is searched in the memory allocation example (Figure 3-9) requires having to code a loop to examine each value in the event queue. Experience with Dalek is identifying which primitive mechanisms are most commonly used and how they are commonly combined. These common combinations will be made available to the user as higher-level mechanisms in the debugger language.

3.5. Extension for Distributed Environments

Event-based debugging has the advantage of pulling together related information that is scattered among different contexts or is only available at separate points in time, which is exactly the case in distributed programs. For example, a message send and a message receipt take place at different times and in different process's contexts. In the dataflow approach, each can be represented by its own event and a higher-level event can represent the completion of a message exchange. As shown by the example in Section 3.2.6, Dalek can be successfully used for debugging pseudo-concurrent simulation programs. These programs can be viewed as repeatable concurrent programs. Their executions are deterministic, but they share many of the characteristics of (true) concurrent programs such as existence of multiple processes, shared global variables, and the need to use synchronization primitives to prevent deadlocks or corruption of shared data.

One potential problem with the dataflow approach is the fact that for distributed systems, time imposes only a partial, rather than a total ordering [Lamp78]. This has profound implications for the semantics of event-based approaches in general. One consequence of the lack of a total order is that, in many cases, it may be impossible for a high-level event to determine which of its constituent events occurred first. Debuggers for parallel programs (e.g., Parasight [Aral88a]) have a global clock and therefore do not have such difficulties. Another consequence is an existential problem: How is a debugger to distinguish between an event that has occurred remotely, but has not yet been *reported* to the debugger by another node, and an event that has simply not yet occurred at that remote node? Or, to phrase it in the terminology of distributed systems, even for a distributed computation that eventually halts, trying to monitor its fairness and safety properties by checking certain conditions for "eventually" and its inverse "never"—even over any finite interval—can require a good deal of patience, since such questions may not be answerable until after the computation is *known* to have terminated.

Using the dataflow approach to debug a distributed program, one debugging process can be associated with each application process. In addition, a master debugging process interacts with the user and the other debugging processes. Each debugging process contains a dataflow engine for part of the dataflow graph. Event recognitions that involve events from only one process can be carried out locally by the corresponding dataflow engine. Such local event recognition on several processes can be carried out concurrently since the tokens and the computations carried out at the nodes are independent. Only those events that depend on lower-level events from more than one process require tokens to be passed between different engines. The overall efficiency of this approach therefore depends on the locality of the dataflow graph. In general, the cost incurred in processing the occurrence of a primitive event through its triggering of a high-level event can be expensive since messages need to be passed. Such costs, and their attendant intrusiveness, seem to be inherent in any event-based scheme that attempts to be interactive versus post-mortem.

In the following sections, we discuss the issues involved in extending Dalek to debug parallel and distributed programs. Such an extension is non-trivial and it increases the complexity of Dalek many times. Many of the assumptions made in designing sequential debuggers become invalid. In practice, the underlying implementation of such debuggers is almost completely different from that of sequential debuggers, although the general principles of information collection and analysis remain unchanged. We have fully implemented a debugger with all the features that we would like to have. However, we have built prototypes to demonstrate the feasibility for some of the techniques we describe. The exact details in extending Dalek to the distributed environments and other potential problems that might occur are left for further investigation.

3.5.1. Architecture of the Debugger

One of the fundamental differences between concurrent and sequential programs is that there are multiple threads of control in concurrent programs. Traditional sequential debuggers have all the data structures and functions that implement the monitoring and controlling of inferior processes built around the assumption that there is only a single context. However, when dealing with concurrent programs, a debugger must maintain the *state* of each individual thread of control, and keep track of the context in which each thread is executing. In other words, for each individual thread, a debugger must maintain separate information regarding its execution behavior, such as the value of the program counter, registers, and local variables.

It is clear that one important design decision needs to be made is: Should the debugger itself be a concurrent program? We believe the answer should be positive. Since each individual thread may execute concurrently, multiple interactions between a debugger and a concurrent program might take place at the same time. A sequential debugger that is time-shared between each thread might induce delays in the inferior's execution and thus might affect its execution behavior. In fact, a debugger should maintain as many separate threads as in the inferior program, with each thread monitoring the execution of a corresponding thread in the inferior program. Furthermore, if an inferior program dynamically creates and terminates threads, the debugger should be able to keep track of that and respond correspondingly.

The design of our concurrent debugger consists of a master debugging process and a number of debugging ***agents***. Each of these debugging agent is responsible for controlling and monitoring the execution of a separate thread in the

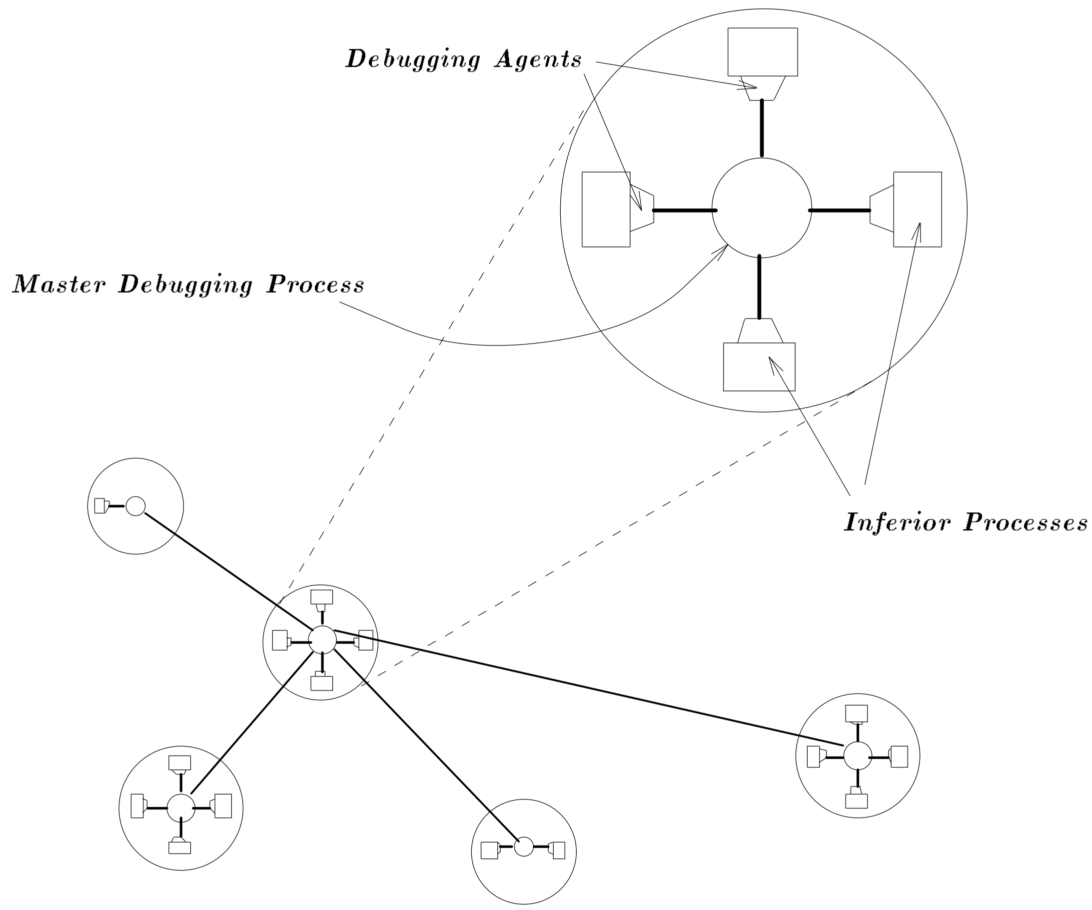

Figure 3-15. Debugging in the distributed environment.

inferior program. Information collected is passed back to the master debugging process for analysis. Depending on the execution environment, one or more master debugging processes might be used. For example, in a distributed environment, it might be more efficient to have one master debugging process running on each physical machine and have those debugging agents running on the same machine report to it. Figure 3-15 shows a distributed program that runs on five different machines, each with different number of inferior processes. Each of the inferior processes has a corresponding debugging agent, which in turns is controlled by a master debugging process. Under this scheme, analysis that involves only local information can be processed locally and thus the amount of information being passed across machine boundaries can be minimized.

3.5.2. Operating System Support

As mentioned in Section 1.2.4, support from the operating system is essential for implementing debuggers, and yet many existing operating systems have very poor support for debugging. Supports for those specific needs required by the implementation of debuggers for concurrent programs are even more inadequate. We have built several experimental concurrent debuggers under the UNIX operating system. Two of the most important problems are discussed here.

The first problem is the sharing of text segment by multiple processes. In UNIX, text segments of all processes are write-protected. If more than one copy of a program is being executed in parallel, the corresponding processes share the same text segment. The only case where the text segment can be written into is when a process is being debugged. Through a special operating system call, a debugging process can insert breakpoint instructions into the inferior process's text segment. When this inferior process hits a breakpoint instruction, control will pass back to the debugging process. Problems arise when one tries to debug a process whose text segment is currently shared by others. A common approach employed by traditional UNIX systems is to disable debugging whenever the text segment is shared. Thus, programs that contain multiple processes executing the same text cannot be debugged, unless multiple copies of the executable file are explicitly created. Some variants of UNIX solve this problem by transparently duplicating the text segment whenever it is written into.

A similar problem under a different context is in debugging programs that contain lightweight tasks or multiple threads. These programs simulate concurrency within a single UNIX process by implementing their own multiprogramming mechanism. Typically they maintain data structures to keep track of the context of each thread or lightweight task and switch between contexts according to some scheduling algorithm. These lightweight tasks obviously share the same text segment because there is only one UNIX process. For the same reason, the operating system does not prohibit the debugging process from inserting breakpoint instructions. However, since the text is shared, any breakpoint instructions will affect all lightweight tasks. In other words, there is no way to set a breakpoint for a particular thread without interfering others. The best that one can do is to examine each thread that hits the breakpoint and resume execution for those threads that are not supposed to stop.

The second problem lies in the inadequacy of UNIX's debugger-inferior communication mechanism:

- The inferior process must be the immediate child of the debugger process. That is, a debugger can only debug processes it creates. If an inferior process forks and creates other processes, which is typical for concurrent programs, the debugger will have no control of these new processes. This restriction has been eliminated by some implementations of UNIX, such as the SUN Operating System. In these UNIX systems, a debugger process can take over control of any specified executing processes. However, when an inferior process forks, the newly created child process immediately starts executing. Thus, by the time the debugger takes control over this new process, an undetermined number of instructions have been executed.

- Synchronization between the debugger and inferior process is carried out by exception handling. The debugger usually suspends itself and wait for signals from the inferior process. When the inferior process hits a breakpoint or any other exception occurs (e.g., division by zero or memory fault), it will be suspended by the operating system, which in turn delivers a signal to the waiting debugger process. In other words, interactions between the debugger and the inferior process is not asynchronous. The debugger has to explicitly make a system call that blocks itself and wait for a signal from the inferior process. Thus, if there are more than one inferior processes, unnecessary delay is introduced. In our implementation, we avoid this problem by using a separate debugger process (agent) to control each individual inferior process. Signals are passed only between each corresponding pair of debugging agent and inferior process. Communication between agents and the master debugger process is by conventional interprocess communication protocols, such as sockets and pipes.

3.5.3. Semantics of the Debugger Language

When there are more than one inferior processes under the control of a debugger, we need to derive ways for programmers to specify which inferior process should be affected by a given debugger command. That is, a programmer should be able to control independently the execution of each individual inferior process, or any subset of inferior processes as a group. Furthermore, the semantics of many debugging actions needs to be redefined. Some of the necessary changes are described in this section.

To illustrate the relationship between a debugging action and multiple inferior processes, consider the insertion of a breakpoint. In the sequential environment, inserting a breakpoint to a certain location causes the entire inferior process to stop when execution arrives at that point. However, when there are more than one inferior processes, a programmer must specify which of the inferior processes should be affected by the breakpoint. Should the breakpoint affect only one process or a subset of processes? Or should it affect all processes that might hit this breakpoint? As pointed out in the previous section, the kind of support provided by an operating system greatly affects how easily each of these options can be implemented. Also, how should the inferior processes react when this breakpoint is hit? Should we suspend only the process that hits the breakpoint while allowing all other processes to run? Or should we suspend all inferior processes? If we need to stop all processes, and if the processes are running in a distributed environment, then the debugger must broadcast a message to notify other processes to stop, possibly using one of the known distributed algorithms to maintain a consistent state [Chan85].

The semantic of the "single-step" command is even more complicated. Dalek allows programmers to single-step the execution of a inferior process through either one machine instruction, one statement, or one function call at a time. In a multi-process environment, interpretation of the single-step command has many possibilities:

(1) Single-step any specified inferior process (or subset of processes) while allowing other active processes to execute uninterrupted.

(2) Stop all processes and single-step only the explicitly specified process(es). A programmer might want to switch between processes and let each process or group of processes execute a few instructions or statements at a time.

(3) Stop all processes and synchronously single-step them all together. That is, in each step every process goes together one step forward.

Chapter 4

Dynamic Linking

In this chapter, we temporarily divert the focus of attention from debugging and describe a new approach to dynamic link/unlink editing [Ho91b]. In the next chapter, we will show how this approach can play a vital part in speeding up the interactions between a debugger and its inferior processes. In addition, this new approach has proved to have many important applications in other areas. We have implemented a prototype link/unlink editor called *dld* [Ho91a] using this approach. *Dld* is a library of functions for C programs running under the UNIX operating system. In this chapter, we describe in detail the design and implementation of *dld*, and discuss its potential applications other than debugging.

4.1. Overview

Many conventional operating systems—such as UNIX, DOS, and VMS—assume that programs are static entities in the sense that construction of a program is completed before its execution. A program's functionality, control structures, number of subroutines, and requirement on library functions are all well defined and do not change once the program begins execution. However, some programming languages, such as LISP and Prolog, take an alternative approach in which they allow new functions to be added during the execution of a program. The assumption that programs are static therefore makes it very difficult to translate these languages directly into native machine code. Instead, they are interpreted by a runtime support system or pseudo-machine, which runs more slowly than native code on a physical machine.

Our approach to program construction allows object modules to be dynamically *defined* or *redefined*, and *added* to or *removed* from a process during its execution. In other existing systems, object modules can at best be dynamically loaded but not removed. Using this approach, the functionalities provided by a program during its execution can therefore change with time or in response to the environment. Thus, this approach retains the efficiency of executing native machine code and adds the flexibility of modifying a program during its execution.

Dld integrates or removes object modules at run time. It differs from other dynamic linkers in that not only can object modules be added to but they can also be removed from an executing process. Furthermore, these modules do not even have to be known or exist when the execution begins. Later in this chapter we describe several applications—program customization, incremental program development, and support for advanced debugging and testing features—that illustrate the usefulness of this dynamic linking approach.

The major cost of dynamic linking is the onetime overhead in reading object modules from disk. The processing time spent on link editing is actually very small. Once the modules have been linked, the executing process runs at nearly the same speed as the equivalent statically linked process. In fact, the only significant drawback of this dynamic linking approach is that its flexibility makes it susceptible to misuse. For example, a process might be corrupted by linking in erroneous code, or careless use of dynamic linking in privileged system programs might create security problems. A later section discusses these drawbacks and possible remedies in more detail.

4.2. Background

This section presents some basic concepts of program compilation and link editing; it also describes the structures of relocatable object files and executable files in general. Further details on these basic concepts can be found in [Beck85]. While the description of these structures and the example given in this section might not directly apply to all operating systems, the underlying principles are all similar.

Most contemporary programming environments do not convert programs written in high-level programming languages directly into executable machine code. Typically, the source of a program is contained in one or more files, or ***source modules***, each of which contains definitions of functions and data structures. These modules are first compiled into ***object modules***, which are then combined together into a single file. This file, known as the ***executable file***, has a well-defined format understood by the operating system and can readily be turned into an executing process.

An object module is the machine code equivalent of its corresponding source module. It contains a ***text*** segment (machine code for functions) and a ***data*** segment (machine representations of global and static variables, string constants, etc.). Throughout this chapter, the term *module* is used whenever there is no ambiguity or need to distinguish between a source module and an object module. When a source module is compiled, the compiler generates an ***object file***, which, in addition to the object module, contains global symbol definitions and information that enables this module to be relocated. This extra information is necessary because global symbols defined in a given module might be referenced by other modules, which need to know the location of these symbols. Furthermore, the compiler or assembler does not know the location in the address space where the module will be loaded when it is combined with other modules. Therefore, relocation information must be recorded with the object file. Related groups of object files of commonly used source modules—such as system services, input/output operations, and mathematical functions—are often combined into archive files, or ***library files***. As a result, users can specify a large number of related object files by giving only one file name.

The activity of loading and relocating object modules into an executable file is generally referred to as ***link editing*** or simply ***linking***. This activity maps each module to a section of the virtual address space, resolves global symbol references across modules, and allocates storage for the global data structures. Each module is then relocated accordingly and the results are written out to a file in the executable format.

Object modules to be linked together can be taken from individual object files or library files. A typical link editor usually handle these two types of files differently. For a simple object file, the object module it contains is always loaded into the executable file. For a library file, since not all modules it contains are generally needed, only those modules defining an unresolved external reference are loaded. Since a module from a library may itself contain references to other modules, loading it may generate additional external references. Therefore, link editors are responsible to search through the library files to ensure all required modules are loaded.

The ability for a linker to automatically select and load only the required modules from a library file is very important: it alleviates the need for a user to keep track of which modules are required. However, the executable file might still contain inaccessible functions or data because even if only one of the functions defined in a module is needed, the linker loads complete the module.

Figure 4-1 shows an example of linking 3 files: `main.o`, `sub.o`, and `libc.a`. `Main.o` and `sub.o` are object files and `libc.a` is a library archive, containing object files `lib1.o`, `lib2.o`, ..., `libn.o`. As shown in the figure, `main.o` contains references to `sub.o` and the function `foo`, which is defined in `lib1.o`. Function `goo`, also defined in `lib1.o`, contains a reference to `hoo` defined in `lib2.o`. Since the smallest unit of linking is an object module, `lib1.o` is loaded completely into the executable file, even though `goo` is actually not referenced at all. Furthermore, `lib2.o` is also loaded because it defines `hoo`, which is referenced by `goo`. The resultant executable file generated will contain 4 modules: `main.o`, `sub.o`, `lib1.o`, and `lib2.o`.

4.3. Dynamic vs. Static Linking

This section describes static linking and its limitations and introduces the concept of dynamic linking. It also points out that existing "dynamic linkers" actually support load-time linking only. They do not allow object modules to be removed

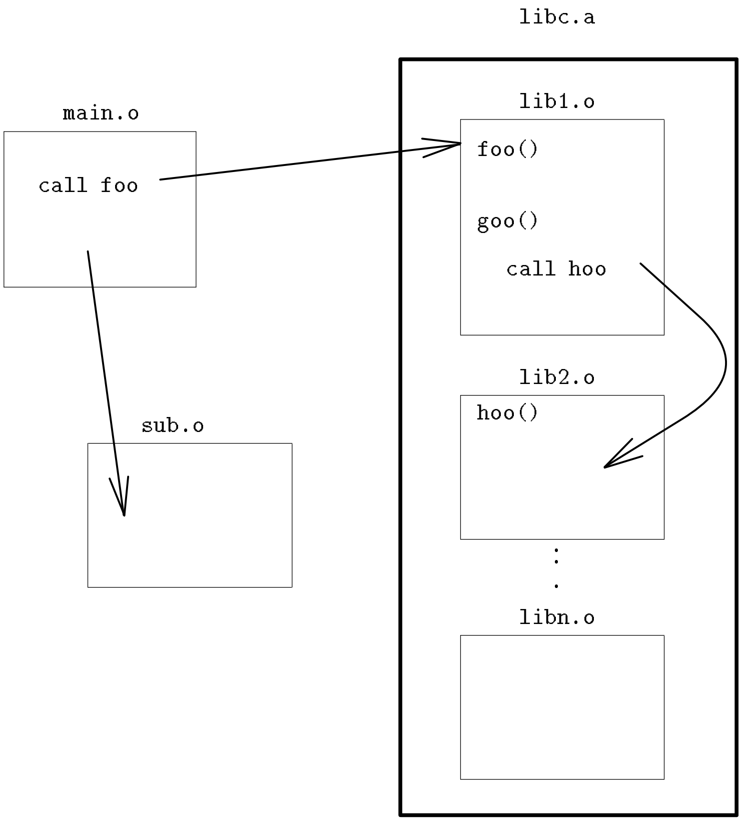

Figure 4-1. Linking of object and library files.

from an executing process, and thus have limited applications.

4.3.1. Static Linking and Its Limitations

Most operating systems assume ***static linking***. That is, the link editing step is carried out only once to produce an executable file, which is loaded directly into memory when it is executed. During the lifetime of an executing process, the (virtual) locations of the text and data segments cannot change. As a result, these operating systems can safely allocate the remaining address space for the stack and dynamic data storage area.

Systems that employ static linking require all global symbols to be well defined at link time. This requirement is a disadvantage because all object and library files must be available during the construction of an executable file. As a result, it is difficult to test and debug portions of a large program incrementally before the whole program is completely written (the best a programmer can do is to write dummy routines for the unfinished part). Furthermore, the entire program has to be relinked if any of the object modules are modified, or if new modules are to be added. Relinking of all object modules can be very time consuming. But unfortunately with static linking, it is unavoidable even in situations where most of the object modules remain unchanged.

These drawbacks make static linking unsuitable for certain applications. For example, a graph plotting program may permit users to specify their own arithmetic functions to be plotted. It would be most convenient if users could define their own functions using the programming language with which they were most familiar, and then incorporate these new functions into the plotting program. However, the use of static linking precludes this obvious approach because the text segment of a program cannot be changed. Instead, most plotting programs incorporate their own special interpreted language in which users define their own functions. Consequently, users are forced to learn a new language, which might not be as powerful as a general-purpose programming language, thus limiting the expressiveness of the users. Furthermore, the design and implementation of an interpreter for such a language increases the complexity and development cost of the originally very simple program. Efficiency of the program is also degraded because, in general, interpreted code executes more slowly than native machine code.

4.3.2. Dynamic Linking

Unlike static linking, ***dynamic linking*** allows a process to add, remove, replace, or relocate object modules within its address space during its execution. In other words, programs are allowed to change. During the lifetime of its execution, a program may have new modules added, old modules removed, or even evolve into a completely different program. For compiled languages, the traditional concept that the code of a program does not change is no longer valid.

Some existing systems support what they call dynamic linking but it is actually load-time linking. For example, in Sun Operating System Version 4, link editing consists of two phases [Ging87]. After the static link phase, the executable file created contains only references to, but not the actual code for the library routines. Integration of these routines with the executable file is carried out by the load-time link phase, during which a linker is called to search and load the missing routines from the library before control is passed to the main procedure. This system requires all global symbols to be declared in the library, though not necessarily defined, during the static link phase. After execution begins and control is passed to the main procedure, no modules can be added or removed. Furthermore, if any external symbol reference cannot be resolved, the program will be aborted. A similar approach is used by HP-UX [Saba90], UNIX System V Release 4 [ATT90], and VMS [Hobb87]. In addition to load-time linking, OS/2 also supports run-time linking. But this scheme requires much operating system support and dynamically linked modules cannot reference any external symbols defined by other modules except the module entry point. Old systems such as MULTICS [Dale68] and languages such as COBOL also support some kind of flexible linking/loading mechanism. However, none of the systems mentioned above allows a program to dynamically modify its functionality during the execution.

One incremental compilation system [Crow87] uses a different approach: it provides library procedures that load additional object modules into the data area of a running process and allows these modules to be executed. This system does not relocate or resolve external symbol references. Instead, it locates every symbol indirectly through an on-line symbol table pointed to by a special reserved register, thus causing a degradation in execution speed. Furthermore, this system requires extensive modification to the compiler, assembler, and the libraries to accomplish this special addressing scheme. While this system allows modules to be added incrementally to an executing process, old modules cannot be removed. Consequently, using this system to implement applications such as interactive program development may not be practical because obsolete modules accumulate and may eventually fill the memory space.

4.4. Dld—A Genuine Dynamic Linker

This section describes the major design and implementation issues of *dld* that provides all the functionalities of a dynamic linking system. The current implementation[1] of *dld* is a collection of library routines that are called by programs written in C running under the UNIX operating system. No modification of the existing compiler or assembler is necessary. The standard UNIX system linker is used only to create the initial executable file. All the dynamic link editing is carried out by invoking routines provided by *dld*. Although *dld* is targeted for C and UNIX, the underlying concept—the use of library functions to build a dynamic linker—is applicable to many other programming languages and operating systems.

4.4.1. General Operations of Dld

Dld's two basic operations are "link" and "unlink". It also provides supporting functions for looking up the addresses of global symbols and entry points of functions.

The link operation is performed by the function `dlink(char *filename)`, where `filename` specifies either a relocatable object file or an object library. If the specified file is a relocatable object file, it is completely loaded into memory. If it is a library file, only those modules defining an unresolved external reference are loaded. Since a module in the library may itself reference other routines in the library, loading it may generate more unresolved external references (as was seen in the example in Figure 4-1). Therefore, a library file is searched repeatedly until a scan through all library members is made without having to load any new modules. Since a UNIX process cannot expand its text segment, `dlink` allocates

---

[1] A number of functions in the current implementation of *dld* are borrowed and modified from the *ld* link editor developed by the GNU project of the Free Software Foundation.

storage for all these new modules from the dynamic data area—the *heap*—using *malloc*, UNIX's memory allocator (assuming the system has writable executable memory). After all modules are loaded, `dlink` resolves as many external references as possible. Note that some symbols might still be undefined at this stage, because the modules defining them have not yet been loaded.

Unlinking a module is simply the reverse of the link operation. The specified module is removed and the memory allocated to it is reclaimed. Additionally, resolution of external references must be undone. For instance, if the module `foo` is to be unlinked, all references to any symbols defined in `foo` from other parts of the program must be marked "undefined". *Dld* provides two functions for unlinking a module: `unlink_by_file(char *filename, int hard)` and `unlink_by_symbol(char *symbol, int hard)`. The first function requires as a parameter the filename corresponding to a module previously linked in by `dlink`, while the second function unlinks the module that defines the specified symbol.

Figure 4-2 shows a simple example illustrating some of the *dld* functions. This program repeatedly reads from the standard input the name of an object file and that of the user function to be executed. It then links in the specified file, executes the named function, and finally removes it from the memory.

On line 13, the function `dld_init(char *filename)` performs the required initialization of the *dld* package. It takes as argument the initial executable file of the program and loads the symbol table information of this file into memory. Line 24 demonstrates how the entry point of a dynamically-linked function can be obtained. The value returned by `get_func(char *func_name)` can later be used as a pointer to the function, as shown in line 27. The predicate function `function_executable_p(char *func_name)` tells whether the specified function

```
1  /* The name of the object file and the function name are read from the
2     stdin.  The named function is invoked through a pointer.  For
3     illustrative purpose the object file is unlinked right after the
4     invocation.
5  */
6  #include <stdio.h>
7  #include "dld.h"
8
9  main (int argc, char **argv) {
10
11     char file_name[80], func_name[80];
12
13     dld_init (argv[0]);
14
15     printf ("object file? ");
16     while (gets(file_name) != NULL) {
17         register void (*func) ();
18
19         dlink (file_name);
20
21         printf ("function name? ");
22         gets(func_name);
23
24         func = (void (*) ()) get_func (func_name);
25
26         if (function_executable_p (func_name))
27             (*func) ();
28
29         unlink_by_symbol (func_name, 1);
30
31         printf ("object file? ");
32     }
33 }
```

Figure 4-2. Simple illustration of the dld functions.

can be safely executed, i.e., whether the execution of this function might lead to referencing any undefined symbols. The precise definition of this predicate function is described later. In this example (line 29), `unlink_by_symbol` is used to remove the new module.

4.4.2. Semantics of Unlink

As seen above, the unlink functions actually take two parameters. The first one is the name of a symbol or file, while the second one is a boolean parameter. When the parameter `hard` is zero (*soft unlink*), the specified module is marked as ***removable*** but it is actually removed from memory only if it is not referenced by any other modules. On the other hand, if the parameter `hard` is non-zero (*hard unlink*), this module is removed from memory unconditionally. Since unlinking a module may leave some remaining removable modules unreferenced, a garbage collector is always called to remove these unreferenced modules. As a special case, library modules are always considered removable and are garbage-collected whenever they are not referenced by other modules. However, users can explicitly request a certain library module to be kept in memory even if it is not referenced.

The semantics of these two unlinking options can best be explained by an example. Consider two modules, A and B, both containing a call to a function defined in C. These modules are linked and removed in the order as shown in Figure 4-3. A is linked in first (a). Then C is also linked (b). When B is linked in, all its external symbol references are resolved successfully because C has already been loaded into memory. Thus, C does not need to be linked in again. Instead, it is shared by both A and B (c).

Now, suppose that A is unlinked, and in order to clean up any module referenced by A, C is being unlinked, too. Should C, which is currently referenced by B, be removed from memory? Both possible semantics are reasonable. If *unlinking* a module means it may be removed if it is not being referenced anymore, C should not be removed (d1) because it is still referenced by B. On the other hand, if *unlinking* literally means removing the specified module, C should be re-

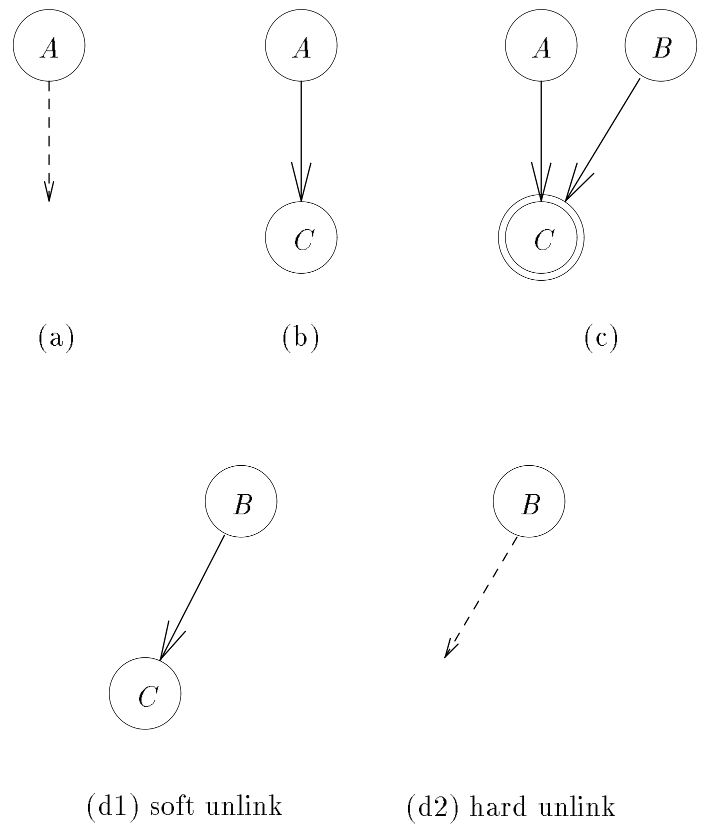

Figure 4-3. Illustration of the two different unlink options.

moved, resulting in B's reference to C becoming unresolved (d2).

Applications might require either or both unlink options. Thus, *dld* supports both options and leaves the decision to the programmer. For example, if the module defining a commonly used function is to be replaced by an enhanced version, a hard unlink is desirable. A new version could then be linked in, and all references to the original function would be modified to point to this new function. On the other hand, some modules might be referenced from many different places in the program. It is not easy for a programmer to keep track whether a module is no longer referenced by others and thus safe to be removed from memory. The programmer might even be unaware of some such references. For example, functions such as *strlen* and *malloc* are referenced by at least a dozen other library functions in the C library of BSD UNIX. Thus, these functions often need to be kept in memory even if the main program does not invoke them explicitly. In these cases, it is desirable to use soft unlink so that the specified module is removed only when it is no longer needed.

4.4.3. Implementation of Unlink

The implementation of the unlink functions is built around a simple garbage collector. For each module x, a reference count records the number of modules that reference any symbol defined in x. If x is explicitly linked in by `dlink` (as opposed to implicit linking in the case for library modules), it is considered being referenced by the process itself, and thus its reference count is at least one. A soft unlink of x causes its reference count to be decremented by one, while a hard unlink simply resets the count to zero. Furthermore, if x's reference count becomes zero, the reference counts of all modules that x references are also decremented by one. When all reference counts have been updated, modules whose reference counts are zero are actually removed by the garbage collector.

In addition to the garbage collector, *dld* maintains data structures to hold information necessary to support unlinking. Corresponding to each module is a list of symbols that it references, a list of symbols that it defines, and a list of relocation instructions (known as an *nlist* in UNIX). This information is needed to update the on-line symbol table in order to reflect the disappearance of the unlinked modules from the program. When a module y is to be removed from memory, its reference list is used to locate those modules whose reference counts should be decremented. Symbols defined by y should be deleted from the symbol table. Other modules that reference any symbol defined by y now contain undefined references. These modules should be "reversely" patched so that instructions pointing to locations previously in y's address space are reset. In short, all *dld*'s data structures have to be set to the state as if y had never been linked into the program.

Finally, side-effects—such as modification of global variables, input/output operations, and allocations of new memory blocks—caused by the execution of any function in a module are not reversed when the module is unlinked. If these side-effects need to be undone, they must be undone explicitly by the programmers.

4.4.4. Deciding if a Function is Executable

Since *dld* allows modules to be added to or removed from an executing process dynamically, some global symbols may not be defined. As a result, an invocation of a function might reference an undefined symbol. To solve this problem, *dld* provides a predicate function `function_executable_p`, which, as shown earlier in Figure 4-2, takes as argument a function name and returns non-zero only if the named function is executable.

A function is *executable* if and only if all its external references have been fully resolved and all functions that it might call are executable. This recursive definition suggests that determining whether a given function is executable is non-trivial. Since each function might invoke other functions, the number of functions that can be reached from the original invocation can be very large. If any of these functions is not executable, the original invocation is also not executable. Consequently, determining if a function is executable could involve the examination of a large number of other functions. For example, the executable file for *gdb*, a popular UNIX debugger, contains 174 modules (object files) and 635 global functions. The efficiency of the algorithm used to implement `function_executable_p` is thus of practical concern. Optimal algorithms with respect to the order of complexity exists. However, in the current implementation of *dld*, we employ a sub-optimal but much simpler solution. This solution has a very small overhead and thus works faster than the optimal solution as long as the number of modules is not too large.

Several methods can be used to speed up this algorithm. First, the *executability* for each function could be actually calculated only once, and the results kept in a table. Subsequent calls to `function_executable_p` would then involve only a table lookup. This table would be invalidated whenever a link or unlink operation is performed, and would not be updated until the next call to `function_executable_p`. As a result, unnecessary recomputation of the table between consecutive link or unlink operations could be avoided. Second, when new modules are added or removed, minimal changes to the table of executable functions should be made. There is no need to rewrite the whole table. Only those functions affected by the changes should be examined. Lastly, although an algorithm that finds out the executability of a given function can be used repeatedly on every defined function, this method is in general very inefficient because a lot of the computation is duplicated. Instead, a different algorithm that finishes all the computation in one single pass could be used.

4.5. Applications of Dld

The flexibility provided by *dld* allows a new style in writing programs using compiled-based programming languages and makes many new kinds of applications possible. In particular, dynamic linking combines the efficiency of executing native machine code with the flexibility of runtime program modification. *Dld* is especially useful for highly interactive programs whose functionalities change in response to their user's input. This section describes several interesting applications of *dld*.

4.5.1. Program Customization

Many sophisticated software packages usually allow some form of customization by the user. Depending on their personal preference, users can specify how these programs should interact with them. Usually these programs provide a group of different options from which users can select those that suit their needs. For example, in the UNIX editor *vi* [Joy86] a user can specify whether the input text should be wrapped around automatically to the next line when input gets past a certain column. As another example, some versions of the UNIX command interpreter *csh* provide an option to log out the user automatically when it has been idle for a user-specified number of minutes.

This method of providing program customization is limited in the sense that users can only choose their preference from the set of pre-defined options. Whatever is not anticipated will not be available. For example, it is not possible to customize *vi* so that when it breaks up a long line, it also right-justifies the current line to a specific column. Likewise, it is not possible to tell *csh* to automatically log out at, say, 5:30pm simply because the need for this functionality was not anticipated. One possible solution is to let the users modify these programs according to their own preference and have separate private copies for each of them. Obviously, this method requires a lot of disk space and makes system software very hard to maintain and upgrade.

Dld provides a better solution by allowing users to add or remove new functions or modify the application programs to suit their own preference. The system then needs to keep only one copy of each program that is loaded with default options. If users are not satisfied with any of the options provided, they can then link in their own routines and tell the program to use these user-defined functions instead of the defaults. Referring to the previous *csh* example, if *dld* is used, a user could perform the following steps to customize the autologout feature:

(1) Use `unlink_by_symbol` to unlink the function that handles autologout.

(2) Modify the way the autologout function interprets its argument. For example, "17:30" means logout at 5:30pm while "17" means logout if *csh* has been idle for 17 minutes. Also make necessary adjustments so that the alarm clock is set correctly.

(3) Compile this new autologout function.

(4) Use `dlink` to link this function into *csh*.

Note that this method requires users to have some knowledge on the implementation of the application programs. Alternatively, the application programs can provide "hooks" or some well-defined, uniform interface to which the users can attach their own customized routines. Users should follow the specified convention when writing their own functions and the application programs may provide a special routine to handle the loading of the user-defined functions. This routine should hide from the users the details of invoking *dld* and of setting up the function entry points.

4.5.2. Incremental Program Development

Since *dld* does not require all source modules to be available when creating an executable file, it makes incremental program development possible. Programmers can start testing their programs before they finish writing the complete program without providing dummy definitions for the missing procedures. As long as the test input data does not cause the program to reference modules that have not yet been linked (i.e., does not invoke a function `f` such that `function_executable_p(f)` returns zero), the execution will proceed smoothly. Programmers can therefore develop programs incrementally and carry out thorough tests on individual modules before working on the next one. Furthermore, when new modules are added incrementally, the dynamic linker needs only to resolve external symbol references related to these new modules. In contrast, if static linking is used, link editing of the whole program has to be started anew. In other words, the use of dynamic linking can in general speed up the program development cycle.

Some systems [Chas87, Kauf88] support incremental program development by providing an interactive interpreter for the underlying programming language. These systems allow programs to be written, tested, and modified interactively. However, interpreted code usually runs more slowly than native machine code. The performance of an interpreter is often unacceptable when executing computationally intensive programs. With dynamic linking, interpreted and (possibly optimized) compiled code can be mixed together. Programmers might use compiled versions of the parts of the program that have been tested, while at the same time they can use interpreted versions of the modules on which they are currently working. The compiled modules are dynamically linked into the interpreter of these systems. These systems also provide the necessary interface for the interactions between the interpreted and compiled code.

As an example, suppose a graphical front end is to be added to an existing chess playing program, which is computationally intensive. During its development, code for this graphical front end is frequently modified, and so it is best run under an interactive interpreter. However, executing the chess playing routines under the interpreter would be unreasonably slow. With *dld*, compiled code for the chess program itself can be dynamically linked into the interpreter, and can be executed at the same speed as if it were statically linked. Thus, *dld* combines the ability of an interpreter to flexibly and completely control the execution of the still developing routines with the high execution efficiency achieved by the optimized machine code.

4.6. Advantages and Disadvantages

The ability to dynamically link and unlink object modules from an executing process provides much flexibility in the construction of a program. This ability allows the building blocks of a program to change or evolve with time: it turns a "program" into a dynamic entity. The functionalities of such a program can be changed in response to its interactions with the environment. As a result, dynamic link editing makes possible many new kinds of applications that would otherwise be very hard to implement.

Systems with similar capabilities have been introduced before. For example, programs written in languages such as LISP or Prolog can load in and execute new functions at any time during their execution. However, with only static linking, these systems have to be built on top of an interpreter or a simulated machine, which always runs many times slower than real machines. Dynamic link editing, on the other hand, combines the high execution speed of native machine code with the flexibility of dynamically modifiable programs.

As a positive side effect, *dld* helps speed up the program development cycle by allowing incremental program construction and on-line maintenance. As the sample applications illustrate, when new modules are added to a program, only part of the program needs to be modified—the rest of the program need not be relinked. Incremental linking can save a considerable amount of time, especially for large programs that often take many minutes to link. Furthermore, the size of the executable files and the disk storage requirements can be reduced because the system can now keep only one copy for the commonly used library routines and have them dynamically linked into the executing process on demand.

Another potential optimization of storage requirement is to allow commonly used library routines to be shared among multiple processes. Many new versions of UNIX [ATT90, Ging87, Letw88, Saba90] already support shared library. Although the current implementation of *dld* allocates storage for the dynamically linked library modules in the private data area of individual process, a simple modification would allow the process to use the shared copy instead.

One easily overlooked problem in implementing *dld* is the effect of instruction cache or instruction pre-fetch mechanism on the behavior of the executing process. When a process dynamically links and unlinks modules during its execution, it is possible that a module is loaded in memory locations that were previously occupied by another modules. Instructions of these already unlinked modules might still reside in the cache or pre-fetch queue and, depending on the architecture of the processor, these stored instructions might not be automatically invalidated or updated by hardware. Thus, invoking the newly loaded module might result in executing the cached or pre-fetched instructions. Implementing *dld* on these kind of processors requires a software cache coherence scheme, such as that proposed by [Kepp91].

The price for the flexibility provided by *dld* is perhaps the overhead in processing time for the link editing and the extra memory required for holding the symbol table and other bookkeeping information. In practice, the time spent in performing both the link and the unlink operations is very small when compared with that spent in executing the program itself. Once the linking is finished, the program can execute at the same speed as if it were statically linked. Although a dynamically linked function must be invoked indirectly through a pointer (as shown in Figure 4-2), this indirection involves only one extra pointer reference. That extra cost is negligible compared with the cost of executing typically hundreds of machine instructions in the body of a function. In addition, preliminary studies show that the vast majority of the time spent in `dlink` is for reading the object files or libraries from disk. In other words, the overhead in resolving external symbol references and maintaining the symbol table is insignificant.

The current implementation of `function_executable_p` is not complete according to the given definition of executability. External references through pointers are not traced. That is, this function will still return non-zero if the named function uses a pointer to indirectly call another function that has already been unlinked. Furthermore, if one external reference of an object module is unresolved, all functions defined in this module are considered unexecutable. This approximation results in a very efficient implementation, albeit conservative.

Like many other powerful tools, there is always a danger that *dld* could be misused. Similar to its static counterpart, a dynamic linker simply combines object modules together and does not provide any extra protection on the existing code against corruption. For example, *dld* does not check if a function to be unlinked is still active (i.e., has a corresponding activation record on the stack). If such a function is unlinked, the executing process might crash when control is passed back to it because the memory originally holding the code for the function might have been garbage collected and reallocated. Also, a function that was executable at one point in time might not be so when other routines are removed. Therefore, the *executability* of a function should in general be verified everytime before it is called, or everytime after modules are added or removed.

Another potential hazard occurs when a dynamically linked routine is erroneous and its invocation could crash the executing process. Thus, programs that allow user customization by means of dynamic linking expose themselves to potential misuse or destruction. In particular, privileged system commands should never allow user modification for the same reason that privileged UNIX commands should not allow shell escapes [Wood85]. Otherwise, an intruder can easily gain unauthorized access to system resources by modifying an existing privileged command. For other application programs, a well-defined, simple interface should be designed for users to hook-in their own functions.

Chapter 5

Implementation of Debuggers

In this chapter, we focus on technical issues in the actual implementation of debuggers. Debuggers are complex programs and their implementations are further complicated by constraints imposed by the underlying hardware and operating systems, limitations in debugging support given by the compilers, and requirements for minimizing their intrusiveness into processes. We describe in general what is involved in implementing a debugger, and techniques required to overcome common implementation problems. In particular, we describe how the concept of dynamic linking (see Chapter 4) helps introduce new techniques to implement some debugging features. In the next chapter, we present a new, layered-approach for implementing debuggers. This approach can make debuggers easier to maintain and more portable to different hardware/software platforms.

In Section 5.1 we describe in general what the essential components of debuggers are, and what kind of support they require from the operating system. Based on our experience in implementing Dalek under the UNIX operating system, we also discuss the inadequacies of UNIX's debugging support. In Section 5.2, we describe the implementation of some of the new debugging features we proposed, including the use of dynamic linking.

5.1. Structure of a Typical Debugger

Despite numerous studies of debugging techniques, very few publications can be found that describe the actual implementation of a debugger. Nevertheless, the implementation of a debugger is a very complex task that requires knowledge of operating systems, compilers, linkers, inter-process communications, interactive programming languages, and user-interfaces.

This section describes the architecture of a typical sequential debugger and the techniques involved in its implementation. Debuggers for concurrent programs are in general more complex, and their structures vary a lot. We will describe implementation issues of some of the more common approaches employed by these debuggers.

5.1.1. Overview

Simply put, a debugger is a program that allows programmers to examine or modify the behavior of another, quite possibly erroneous, program while it is executing. Typically, a debugger contains the following components:

- **Interface to the inferior process**. This includes code that creates and terminates an inferior process, suspends and resumes its execution, inserts and deletes breakpoints, writes to and reads from its memory image or registers, and also handles signals and exceptions.
- **Manipulation of the inferior process's memory image**. To collect information about the inferior's execution status and locate local variables or func-

tion parameters, a debugger must understand the layout of the activation records and parameter passing mechanism, which are usually system- and compiler-dependent.

- **Decoding of the executable and core files**. Text, data, and symbol tables of a program are put together in the executable file according to a system-dependent format. Likewise, when a process "dumps core", the format of the core file is also system-dependent. A debugger must understand these formats and provide code to read and convert the recorded information into internal representation.
- **General bookkeeping**. A debugger must maintain a symbol table, which is a data structure that holds information about all symbols, local and global, defined in the inferior program. Typical information associated with each symbol includes size, type, layout of records, scope, and address. Other information that must be recorded and constantly updated are, for example, the current execution status of the inferior process, breakpoints that have been inserted, executing environments, and mapping of positions in the source program (such as line numbers) to locations in the text segment (address of machine instructions).
- **Information analysis**. The raw data obtained from the inferior process must be processed, analyzed, and converted into information that is useful for programmers to locate errors.. In the case of Dalek, we use our dataflow analysis mechanism to generate high-level description of the inferior process's behavior.
- **Command language interpreter**. Typical debuggers provide many different debugging features and thus the set of commands controlling their operations is large enough that a command language interpreter is required. Fur-

thermore, in the case of Dalek whose command language contains a general-purpose programming language, a full language interpreter is used to parse its debugging language and execute the specified actions accordingly.

- **User interface**. In the simple case, this involves a command line editor that allows programmers to enter and edit commands to be passed to the command interpreter. The interface should also present information to programmers in a nice comprehensible way. For example, variables in their internal formats should be converted back to the corresponding source-level representations before they are printed, and instructions should be output in mnemonics. More sophisticated user interfaces might include a graphical input and display.

5.1.2. Interface to the Inferior Process

Code that handles the direct interaction with the inferior process is the lowest level and the most system-dependent part of a debugger. Since processes are protected by the operating system from accessing each others address space, interactions between the debugger and inferior process must be handled by special operating system services. As a result, the functionalities and effectiveness of a debugger depends highly on the kind of debugging services provided by an operating system, and how sophisticated they are. This part of the debugger is basically an interface to the debugger services provided by the underlying operating system.

Four types of support from an operating system are essential to the implementation of a debugger. First, a debugger must be able to specify the process to be debugged and take over control from it. Second, a debugger must be able to read data from and write data to the inferior process. Third, a debugger must be able to control the execution (such as suspending or resuming execution) of the inferior process. Lastly, a debugger must be able to intercept all interrupts or signals to be delivered to the inferior process, and handle all exceptions generated by the execution of the inferior process.

5.1.2.1. Taking Over Control

A debugger must be able to specify which process it is going to debug. Normally, an operating system verifies the access privileges of a debugging process and then puts the specified process in a special "being debugged" state. In other words, this process is turned into an inferior process under the control of the debugger, which is now granted permission to invoke system calls that support debugging.

Many UNIX systems allow debugging only between processes that are direct parent and child. Therefore, a process can be debugged only when it is directly created by a debugger. That is, processes that are already executing and processes created by an inferior processes (i.e., grandchildren of the debugger) cannot be debugged. In some implementations of UNIX, such as the Sun Operating Systems, this restriction has been eliminated and a debugger can take of control of any process as long as they are owned by the same user.

5.1.2.2. Accessing Address Space

A debugger needs to read from the inferior process's address space to find out contents of variables, function parameters, or even the machine instructions loaded into memory. It also needs to obtain content of the registers such as the program counter, stack pointer, or frame pointer. Occasionally, it might need statistics about the execution of the inferior process maintained by the operating system, such as the system resources consumed. Likewise, a debugger needs to write onto the inferior process's address space and register pool so that it can modify variables, insert breakpoints, or alter the execution path by changing the program counter.

Typically, a system call is provided for copying a block of memory from the inferior process's address space to the debugger process's address space, or vice versa. Thus, the efficiency of the implementation of such system calls and the frequency of their invocations directly affect the overall performance of a debugger. In many UNIX systems, each call to read data from the inferior process involves two context switches: the first switches from the debugger to the inferior process to read the data, and the second switches back to the debugger to write the data into its memory buffer. Such a system call is therefore very expensive. Furthermore, many UNIX systems impose limits on the size and address boundaries of the memory block to be copied. As a result, debuggers are forced to issue more system calls and hence their performance further degrade. Fortunately, a new implementation has been proposed to eliminate these unnecessary context switches and size limits [Kill84] (see Section 2.4).

An even better mechanism is to eliminate copying of data entirely. By negotiating with its memory management system, an operating system can directly map sections of a debugger's address space onto the inferior's address space so that they share the same physical memory page. In other words, the operating system allows sections of the memory image of the inferior process to be shared by the debugger. Once such sharing is established, the debugger accesses these memory blocks as fast as its own address space.

5.1.2.3. Controlling Execution

The kind of control a debugger needs over the inferior process includes suspension of execution, resumption of execution, and alteration of the execution path. A debugger can suspend the inferior's execution either asynchronously by sending a signal to it, or synchronously by pre-inserting breakpoints. Also, ***single-stepping***, which is the special case of resuming the inferior's execution for just a single machine instruction, often receives additional hardware or operating systems support.

Whenever a signal is delivered to an inferior process, or an exception is raised during the inferior's execution, many operating systems suspend the inferior process and notify the debugger. Thus, to asynchronously suspend an inferior process, a debugger can simply send a signal to it. We will describe the signal handling mechanism in more detail in the next section.

With breakpoints, which are synchronous, suspending the inferior process is a little bit different. Many processors provide a special breakpoint instruction. When a process executes such an instruction, an exception is raised. If that process is currently being debugged, its execution will be suspended and the controlling debugging process will be notified. Hence, insertion of a breakpoint at a given location is actually implemented by replacing the corresponding machine instruction at that point with the breakpoint instruction. Deletion of breakpoints is done by restoring the corresponding machine instructions that were replaced. Note that a debugger has to record all replaced instructions and their corresponding locations for later restorations.

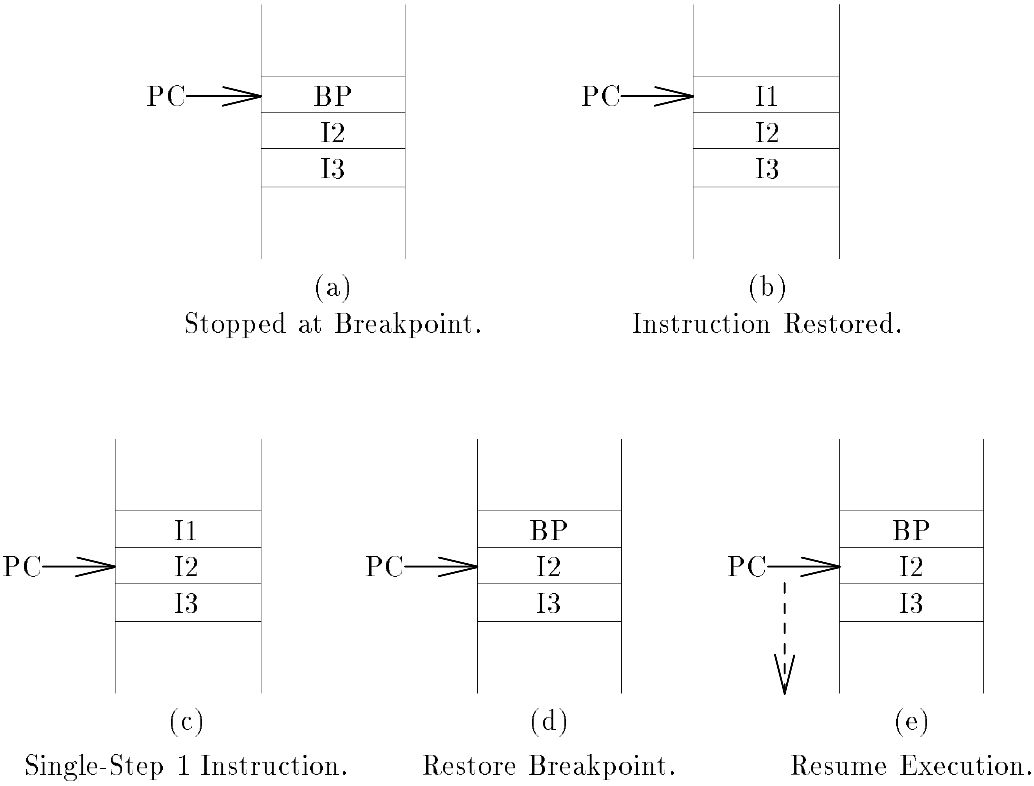

Figure 5-1. Resuming execution from a breakpoint.

Programmers are given the illusion that breakpoints are inserted in between machine instructions without modifying them. This is achieved by transparently restoring the original instruction when the inferior process resumes execution after stopping at a breakpoint. However, if the breakpoint needs to stay in position after the inferior process has resumed execution, as is often the case, the debugger must re-insert the breakpoint immediately after the replaced instruction is executed. As shown in Figure 5-1, this mechanism usually involves five steps:

(1) The inferior process stops at the breakpoint (a).

(2) The replaced machine instruction is restored (b).

(3) The inferior process executes this instruction and stops right after that (c).

(4) The breakpoint is re-inserted (d).

(5) The inferior resumes normal execution (e).

From the above description, we see that an inferior process needs to single-step through an instruction when it resumes execution after stopping at a breakpoint. Single-stepping is also useful for close examination of how the state of an inferior process changes when it goes through a number of instructions. Many processors provide hardware support for single-stepping. In these machines, the status register has a designated bit that, when set, causes a trap to occur after one machine instruction is executed. Implementation of single-stepping on these machines is thus very simple. For those machines that do not provide such support, single-stepping has to be implemented by setting a breakpoint at the next instruction and then resuming the inferior process. Furthermore, if the instruction to be executed is a branch or procedure call, breakpoints have to be set at all possible locations where the branch might reach, and then later removed.

Changing the execution path of an inferior process is relatively simple. A new address is over-written onto the program counter. When the inferior process resumes execution, it will execute the instruction at this new address. Usually, this feature is used to skip over a number of statements inside a procedure, or to force a procedure to return immediately without executing the rest of the statements. However, if the program counter is set to a position outside the boundaries of the function that is currently executing, the debugger is responsible for setting up all the registers and stack content so that execution of the inferior process can proceed smoothly. Options should be provided to either replace the current activation record by that of the new function, or stack a new activation record on top of the current one.

5.1.2.4. Signal Handling

In many operating systems, such as UNIX and Mach [Blac88], debugging and inferior processes synchronize their execution by an exception or signal handling mechanism. Usually, whenever an inferior process receives a signal or raises an exception, its execution will be suspended by the operating system, which in turn notifies the controlling debugging process by sending it a signal. On the other hand, the debugging process synchronizes its execution with the inferior process by voluntarily suspends itself until it is awakened by the signal from the operating system indicating the inferior process's suspension.

Since all signals and exceptions are captured by the debugging process, the inferior process will never see them. The operating system must then provide an alternate way for delivering signals to inferior processes. Usually this is achieved by providing a system call used only by debugging processes. Signals sent by this special system call are delivered to the inferior process without causing it to suspend. As a consequence, a debugger can use this signal handling mechanism to monitor the delivery of signals to the inferior process. When a debugging process receives a signal that is re-routed by the operating system from the inferior process, it can choose to either block or forward the signal to the inferior process. In the latter case, the debugger might even deliver a different signal to the inferior process. In the case of UNIX, for example, the signal delivered by the debugger to the inferior process can be specified as an optional argument in the system call to resume the inferior's execution.

5.1.2.5. Other Debugging Support

Some functionalities of a debugger cannot be implemented efficiently without special support by the underlying operating system or hardware. Two examples of such functionalities are described in this section.

The first example is data ***watchpoints***. In contrast to a breakpoint, which is set in the text segment of an inferior process, a data watchpoint is set in the inferior's data segment or stack. It specifies a memory location that, when accessed or modified, causes the inferior process to suspend and the corresponding debugging process to be notified. Data watchpoints are extremely useful for monitoring data structures that are being accessed from many different parts of a program, or can be accessed via many different paths (e.g., using pointers). Without data watchpoints, a debugger might have to set a breakpoint at every single part of the program that accesses such data structures. Very often, there may be too many such locations in the program to make this method practical. Furthermore, if a program contains memory access via pointers, and if any of these pointers is corrupted, then potentially every single memory access through a pointer might modify the interested data structures. In other words, a debugger might have to single-step the inferior process at every single instruction and check if the data structures have been modified.

A reasonably efficient implementation of data watchpoints requires hardware support. Any software method that verifies if a given memory location has been modified after executing every instruction slows down the execution of the inferior process by orders of magnitude. Some microprocessors, such as the Intel 80386 [Inte86], contains four watchpoint registers. Each memory address to be accessed is compared with these registers and, if it matches with any one of them, an interrupt will be generated. Unfortunately, no hardware implementation supports an unlimited number of watchpoints.

The second example is operating system support for suspending an inferior process whenever it issues a system call. System calls are functions that execute outside of a process's address space, and their implementations are transparent to the process. Therefore, to find out the effect of a system call on the states of an inferior process, a programmer may want to set a breakpoint at the system call, record the state of the inferior process, step through the system call, and then examine how the state has changed.

Furthermore, if a programmer needs to find out how an inferior process affects the system status, such as what files it is opening and what is being written on these files, he or she must set breakpoints at the corresponding system calls and examines the arguments being passed to them. However, system calls are often invoked from various library functions and in general a programmer does not know what system calls are made by which library function. Thus, if an operating system does not support suspension of an inferior process whenever it issues a system call, the only way to locate all system calls in the inferior process is to scan through its entire text segment and search for all system call instructions. This method is not only slow, but also incomplete. For if an inferior process contains self-modifying code or dynamically loads in new instructions, the debugger will have no way to find out when a system call is being issued. Thus, such a debugging feature must rely on operating system support.

5.1.3. Manipulation of the Memory Image

Having the ability to read from or write to the memory image of the inferior process, a debugger must also understand the organization of data structures in that memory image. It has to know which part of the memory corresponds to which data structures.

Locations of global variables and definitions of records are supplied by the symbol tables associated with the executable file. So a debugger can simply consult these symbol tables when it needs to access these structures. However, locations of local variables and function arguments vary while the inferior process executes. Hence, a debugger must have knowledge of the organization of the stack and activation records, and it has to traverse the stack when locating such variables.

Assignment of hardware registers as the program counter, stack pointer, and frame pointer differs from processor to processor. Also, the use of the stack and the structure of activation records are operating system-dependent. Furthermore, each programming language, and sometimes different implementations of the same programming language, has its own parameter passing method. In other words, the knowledge required to understand the memory image of an inferior process is very specific for any given compiler implementation, operating system, and hardware. As a result, debuggers that work on more than one software and hardware configurations must have all the specific information for each configuration loaded. A well-designed debugger should have such configuration-dependent parts isolated from the rest of the implementation so that the general debugging mechanism can be shared.

5.1.4. Decoding the Executable File

Strictly speaking, an executable file should contain only the machine instructions and initialized data of a program, as well as information indicating how a process should be created for this file, such as the initial size of the data area, the location of the first instruction to be executed, and if any shared-libraries need to be mapped. However, it is the convention of many operating systems to attach debugging information of a program, usually known as *symbol table*, to its corresponding executable file. The exact format of how all these components are recorded in an executable file is somewhat arbitrarily defined by the underlying operating system. Thus, decoding the executable file is another part of a debugger that is system-dependent.

The debugging information recorded in an executable file is usually generated by the compiler and it describes how each part of the address space of the inferior process should be mapped to the source program. Corresponding to the data segments, this information contains definitions of types and layouts of data structures, location of each global variables and records, and offsets of local variables and function parameters from the frame pointer. In terms of the text segment, it contains mapping between statements in the source program and the actual memory locations of the corresponding machine instructions.

5.1.5. General Bookkeeping

During its execution, a debugger has to maintain fairly extensive data structures to record the large amount of information about the inferior process and its execution status, and to allow frequent updates and retrievals. The most important of all is the symbol table loaded in from the executable file. As described in the previous section, a symbol table contains information on types and locations of all symbols, and the mappings between machine instructions and statements in the source program. Every time a debugger accesses a variable in the inferior's address space, it has to lookup in the symbol table the location, size, and type of the variable. Likewise, conversion between memory address of machine instructions and the corresponding locations in the source program has to be performed frequently because the debugger communicates with programmers at the source language level but it interacts with the inferior process in terms of memory addresses. Symbol tables are usually recorded in the executable file in compact form, but internally a debugger must record them in data structures that allow fast information retrieval. Furthermore, if an inferior process dynamically loads or removes functions from its address space during its execution, the data structures employed by the debugger have to be flexible enough so that the symbol table can be updated accordingly.

Other than the symbol table, a debugger has to keep track of the execution status of the inferior process. This includes the current function or statement being executed, the current stack of function calls, how each kind of signal should be handled, locations of all breakpoints and watchpoints, and the entire execution environment.

5.1.6. Information Analysis

Having the ability to monitor the execution of an inferior process, a debugger must provide tools for analyzing the information collected. Most existing debuggers for sequential programs do not have any tool for information analysis, but almost all debuggers for distributed programs provide various analytical tools using different approaches. As described in great detail in Chapter 3, we proposed to use the dataflow approach to perform such analysis.

5.1.7. Command Language Interpreter

Since a debugger has so many features and can perform so many different operations in monitoring the inferior process, the set of commands controlling the debugger is usually very rich. Most existing sequential debuggers have relatively simple command syntax and each debugging operation can be specified in one line. Command interpreters for these debuggers are usually implemented in an arbitrary way. Recall, however, that debuggers such as Dalek have rich command languages that include a general-purpose programming language; their implementations therefore require a full implementation of a programming language interpreter. Dalek's implementation includes a parser, a dataflow engine to handle events, processing units that actually performs the debugging actions, storage of source program (i.e., definitions of events, breakpoint commands, etc.), allocation of storage for global variables, dynamic storage allocation of local variables, handling of parameters passing in function call, and many other necessary functionalities of a language interpreter.

Techniques for implementing language interpreters can be found in numerous publications [Aho86, Mak91] and are not within the scope of this dissertation. However, the unique operating environment of Dalek's command interpreter creates an implementation problem that merits special attention. This problem is the result of the fact that the debugger language works on two entirely different address spaces. Recall from Chapter 3 that Dalek allows variables of the debugging code and variables of the inferior process to be mixed in the same expression. In other words, Dalek's statements and functions are being executed under two different contexts. As a result, the same Dalek function might give different results depending on the current scope of the inferior program. For example, suppose a Dalek statement contains a reference to an inferior variable `x`, and this inferior program contains both global and local definitions of `x`, then this same Dalek statement may end up accessing different `x` depending on the inferior function that is currently on the top of the execution stack. In our current implementation, scope rule for accessing variables in the inferior process is evaluated from the viewpoint of the function that is currently being executed, i.e., bindings of inferior variables are dynamic. Furthermore, a Dalek function can change the scope of the inferior process by walking up or down the stack of activation records, and thus can access variables that are outside of the current scope.

5.1.8. User Interface

Although the user interface of a debugger does not perform any *actual* debugging function, it is the only part of the debugger that directly interacts with a programmer, and it directly affects how well a programmer can use it. A user interface performs a two-way communication between the programmer and the debugger. It allows a programmer to enter his or her instructions to the debugger easily, and it also displays all output of the debugger in an easily comprehensible manner.

Undoubtedly, it is the expressiveness of the debugger command language that determines how easily a debugger can be used. Other than that, a user interface can provide other aids to assist programmers in entering their commands. For example, a debugger may provide hooks to allow its commands to be input and edited by a programmer-specified text editor. In the case of Dalek, programmers can use their favorite text editors to prepare definitions of events or breakpoint commands and then send them to Dalek all at once. When reading line-oriented, interactive commands from the keyboard, a debugger may provide a command-line editor [Fox89] so that programmers can correct any typing mistake easily and thus minimize the number of keystrokes.

More sophisticated user interfaces often provide graphical input/output facilities. For example, a debugger may use multiple windows with one for showing the text of the source program, one for reading commands and printing output, and one for normal input/output of the inferior program [Adam86, Russ91a]. Furthermore, it may use a pointing device such as a mouse to specify locations of breakpoints, associate blocks of command text from the editor window with locations in the source program, or define relations between low-level and high-level events. Some debuggers can automatically generate graphical representations of data structures to help programmers visualize the information [Shim91].

5.1.9. Concurrent Programs

Most of the debuggers for concurrent programs are still in experimental states and their implementations vary a lot because of the differences in the approaches they use. As described Chapter 2, most debuggers for parallel or distributed programs generally employ a postmortem approach where execution of the inferior processes are recorded first and analyzed later. Therefore, these debuggers must be able to monitor or intercept any interprocess communication and synchronization activity between these inferior processes. Furthermore, to correctly reproduce the behavior of a concurrent program, the relative-ordering of execution of the inferior processes must also be recorded, and some kind of synchronization algorithms must be used to obtain such ordering information.

5.2. Advanced Debugging Features

In this section, we describe new techniques to implement a number of debugging functions that greatly improve the efficiency of a debugger. Based on these new techniques, we also developed several new, advanced debugging features. Section 5.2.1 describes a new, more efficient way to implement single-stepping at the source language level, as well as a new definition of its semantics. Section 5.2.1 describes a series of new debugging features that build around the concept of dynamic linking.

5.2.1. Source-Level Single-Stepping

Traditional techniques employed to implement single-stepping at the source program level is very slow and inefficient. In this section, we describe the inadequacies of these techniques, and point out the flaws in the traditional concept of single-stepping. By correcting the semantics of source-level single-stepping, we propose a new implementation that is much more efficient.

5.2.1.1. Traditional Implementation

To allow debugging at the source language level, one important requirement is that a debugger must be able to map addresses of machine instructions to the corresponding locations in the source program, and vice versa. Such mapping information is generated by compilers and is incorporated into the symbol table sections of executable files. Usually, each statement in the source program is translated into a number of machine instructions. Thus, only the address of the first of these instructions and the line number of the corresponding statement are recorded as a matching pair. Note that mappings between line numbers and addresses are not one-to-one because a compiler might rearrange the machine instructions, and as a result, instructions corresponding to a single statement might be separated into several groups in different locations. In these cases, extra pairs of line numbers and addresses are generated.

When a programmer sets a breakpoint at a certain line in the source program, the debugger consults the mapping information given by the compiler, finds out the address of the corresponding machine instruction, and replaces it by the breakpoint instruction. Likewise, when the inferior process stops, the debugger reads the value of the program counter, converts it into line number, and then notifies the programmer of the line in the source program at which the inferior process stopped.

Although many operating systems and hardware provide support for single-stepping at the machine instruction level, single-stepping at the source level has to be implemented in software by debuggers. Many debuggers, such as *gdb* [Stal91b], *dbx* [Adam86, DBX83], *sdb* [Kats79], and *DEBUG* [DEC88], implement source-level single-stepping in an very inefficient way. The inferior process is single-stepped at the machine instruction level, and after every instruction, the value of the program counter is examined. These operations are repeated until the program counter points to a location corresponding to a source line other than the previous one. As a result, single-stepping through statements that are translated to a large number of instructions is extremely slow.

Ideally, source-level single-step should be implemented by setting a breakpoint at the next source line and then resuming the inferior's execution. However, the current statement might contain branches or function calls and thus, after executing this statement, the inferior process might not stop at the next line. That is why the single-instruction stepping method described above is used.

If a debugger knows all the possible locations to where the inferior process might go after executing the current statement, it can set breakpoints at all such locations and then resume the inferior execution. Unfortunately, such information is not provided by most compilers. Alternatively, a debugger can find out the possible branch locations by decoding the machine instructions to be executed [Russ91b]. However, this approach makes the debugger more complicated and it still requires a considerable amount of processing for each single-step.

5.2.1.2. Redefining the Semantics of Single-Step

Before we describe the proposed methods for implementing fast single-stepping, we would like to point out that the existing scheme of mapping instruction addresses with line numbers is a wrong abstraction. Modern programming languages are usually block-structured and have a free format syntax. There may be more than one statement in one source line, and a single statement may span a number of lines. The basic syntactic structures of these languages are expressions, statements, and blocks, but not line numbers. Thus, using line numbers as units for setting breakpoints and single-stepping is misleading.[1]

Using statements and expressions as the unit of execution, three options of single-stepping should be provided:

```
while (i < 100) {
    a[i] = b[i];
    ¹if (end_of_line(b[i])) {
        ²putc ('\n');
        break;
    } else
        ³putc (a[i]);
    ⁴i++;
}⁵
```

Figure 5-2. Illustration of step-in, step-through, and step-out.

---

[1]The use of line numbers to reference program statements is probably descended from line-oriented programming languages such as FORTRAN and COBOL.

- **Step through**—The inferior process is allowed to execute one statement or expression and is then stopped. If the current statement is a compound statement, then the entire compound statement is executed before the inferior is stopped.
- **Step in**—If the current statement is a compound statement, or if it is an expression containing a function call, then the inferior process is allowed to execute until it reaches the first statement within the block or function respectively.
- **Step out**—The inferior process is allowed to execute until it finishes the current block or function.

Figure 5-2 shows an example that illustrates the semantics of single-steps. Suppose an inferior process is stopped before the `if` statement at position 1. A step-through from position 1 causes the inferior process to finish executing the entire `if` statement and to stop at either position 4 or 5, depending on whether `end_of_line(b[i])` returns zero. A step-in from position 1 causes the inferior process to execute and stop either inside the function `end_of_line` or at position 2 or 3. Lastly, a step-out from position 1 cause the inferior process to execute until control is passed out of the current block, i.e., after the loop at position 5.

Most existing debuggers for compiled programs (e.g., *dbx*, *gdb*, and *ups*) provide similar support for these kinds of single-stepping only at the function call level. For example, in *gdb*, the `next` command steps through one source line and even if there are function calls on that line, it does not stop inside of them (step through). On the other hand, the `step` command steps through one source line and, if there are function calls on that line, stops right before executing the first statement in the first function call on that line (step in). Lastly, the `finish` command causes the inferior to resume execution until the end of the current function (step out) [Stal91b]. Nevertheless, none of these debuggers generalize these operations to the statement and expression level. Implementation of these three kinds of single-stepping would have been easier in an interpreted environment, because there is no need to map machine instructions back to corresponding source lines. However, this has not been done in many popular interpreted environments, such as Saber-C [Kauf88], and Lightspeed Pascal [Bore88]. The only common debuggers that support single-stepping at the expression level are those for languages such as LISP [Wile84] and Smalltalk [Digi89]. The only structures in LISP are S-expressions and atoms, and so a LISP debugger can only single step at the S-expression level. Although a programmer can put several S-expressions on the same line, hardly any LISP debugger provides functions to step through all S-expressions on the same line. In fact, the concept of line number is seldom used in LISP. Likewise, every operation in Smalltalk is carried out by message passing between objects. So a Smalltalk debugger that captures the messages passed can step through execution of a Smalltalk program in any granularity.

5.2.1.3. Implementation of Fast Single-Step

To implement this new single-step mechanism, a debugger must have knowledge of how control is passed from one statement to another. As described in the previous section, when such control flow information is lacking, debuggers are forced to use very inefficient methods to find out where control will be passed after executing a statement. Therefore, in our implementation of fast single-step, we modified a compiler so that it incorporates a flow graph into the symbol table.

Each program statement is given a unique statement identification number (SID). This SID is used instead of line number. Associated with each statement are two lists of SID's. The first list contains SID's of all statements that might be reached after the current statement is executed. The second list is for compound statements or expressions that contain function calls. It records SID's of all the statements that might be reached if the current statement is stepped in. All this information is generated by the compiler and is embedded into the symbol table of the executable file. Given this information, the debugger can reconstruct complete flow graph of a program.

To step through a given statement, breakpoints are set at all statements shown on the first list, and then the inferior process is resumed. To step into a compound statement or function call, breakpoints are set at those statements shown on the second list. To step out of the current block is the same as stepping through the compound statement that defines the current block. We can obtain the required list of possible destinations by traversing backward in the flow graph. As a result, regardless of the number of instructions expanded from a program statement, the overhead of single-stepping remains very small and the inferior process can execute this statement uninterrupted.

Line numbers are replaced by the SID's to map with instruction addresses. A second table is generated for mapping SID's to locations in the source program. Since there might be more than one statements on the same source line, locations in the source program should be specified by both the line and column numbers.

We have modified the GNU C compiler [Stal91a] to generate the SID tables as described above. To minimize unnecessary changes to the compiler, linker, and debugger, control flow information is written into a separate file rather than embedded into the symbol table. Furthermore, we simplify the implementation by omitting column numbers in the mappings between SID's and locations in the source program. This simplification allows minimal modifications to be made. However, ambiguities do arise because more than one statement may be mapped to the same line. Single-step through statements or expressions on the same line confuses the debugger and causes it to recognize only the first statement and ignore the remaining statements on the same line. This deficiency can easily be solved if column numbers are added.

5.2.2. Debugging with Dynamic Linking

As stated in Chapter 4, the concept of dynamic linking brings in a new range of debugging features. When dynamic linking is used by an inferior process, its modules can be tested, debugged, and corrected separately while it is executing. Thus, unnecessary compilations and linkings are eliminated. When a debugger itself uses dynamic linking, new debugging features can be built that can greatly improve the efficiency of the debugger.

5.2.2.1. Within Inferior Process

Dynamic linking allows components of a program to be replaced at run time. Therefore, if an inferior process already has the dynamic linking package loaded, a debugger can be used as an interactive interface to control its reconstruction. For example, a debugger can invoke dynamic linking functions in the inferior process's address space to remove the erroneous routines. These routines can then be fixed, recompiled, and linked back into the inferior process. Furthermore, new debugging routines can be linked into the inferior process on demand. This feature is particularly useful when the need for such debugging routines is not realized until the inferior process has been executing for a long time and finally arrives at some state of critical importance for debugging. Additionally, many unnecessary compilations and linkings are eliminated. Without dynamic linking, every time a bug is fixed or new routines have to be added, all modified modules have to be compiled again and the entire program has to be linked.

The above debugging features are also applicable to programs that are not designed to use dynamic linking. To debug these programs, a dummy inferior process is first created. This process contains nothing but the dynamic linking code. A debugger can then alter the execution of this dummy inferior process and make it dynamically link in object modules of the program to be debugged. The next step is to invoke the "main" function of this program. Simply put, we have created an inferior process that contains both the program to be debugged and the dynamic linking functions. Erroneous modules of this program can then be replaced and new modules can be added in dynamically as described in the previous paragraph.

This method of using a dummy inferior process as a bootstrap for loading other modules can also be used to debug unfinished fragments of a program. For example, a programmer might want to debug and test the symbol table routines of a compiler before moving on to finish the rest of the program. S/he might set up a debugger for a dummy inferior process as described above and link in the symbol table routines. The programmer could then set breakpoints, examine or modify program variables, or single step through the execution of statements in these functions in the same way as debugging a complete program.

5.2.2.2. Within Debugging Process

One of the important debugging features achieved by using dynamic linking is the implementation of fast conditional breakpoints. A conditional breakpoint is a breakpoint to which an expression is attached. Typically, conditional breakpoints are implemented in the same way as ordinary breakpoints by inserting breakpoint instructions at the specified locations. However, when a conditional breakpoint is hit, the debugger suspends the inferior process and evaluates the associated expression. If this expression turns out to be true, the debugger will handle this breakpoint in the same way as an ordinary breakpoint (e.g., notify the programmer and prompt for the next command). Otherwise, it resumes the inferior process as if no breakpoint had been hit.

Such an implementation of conditional breakpoints is very inefficient. As pointed out in Section 5.1.2.2, data transfer between the debugger and inferior process is an expensive operation. Hence, insertions and removals of breakpoint instructions require considerable overhead. In addition to this overhead, a conditional breakpoint requires the evaluation of the associated expression. This expression may be very complicated and involve many variables in the inferior process. Overhead of locating these variables and copying their values over to the debugging process is thus very high. Furthermore, conditional breakpoints are usually set at frequently executed statements, and the conditions are defined so that uninteresting cases are silently ignored. For example, to stop the inferior process after it has executed a loop for a thousand times, a programmer can set a conditional breakpoint whose condition is true only when the loop index is greater then one thousand. In other words, the overhead for processing a conditional breakpoint is repeated a thousand times.

With dynamic linking, a debugger can generate code that evaluates the conditional expression and dynamically link it into the inferior process [Kess90]. This code can be loaded from pre-defined debugging functions or written by programmers and compiled separately. Instead of using a breakpoint instruction, a simple jump instruction is put into the location of the breakpoint. When execution arrives at this location, control will be passed to the inserted code and the conditional expression will be evaluated. The debugger should set up in the inserted code that if the specified condition is not satisfied, the instruction replaced by the jump instruction is executed and control is passed back to the location of the breakpoint. On the other hand, if the condition is satisfied, the inferior process is suspended and control is passed back to the debugger. By cleverly rearranging and relocating a few machine instructions, the insertion of the branch to and from the inserted code becomes transparent to the original execution. Since many unnecessary context switches and data transfers between the debugger and the inferior are eliminated, the efficiency of interactive debugging is greatly improved. Furthermore, evaluation of the conditional expression is done by native machine code, which is orders of magnitude faster than that by the debugger's command interpreter.

An alternative way dynamic linking can be used to speed up the interaction between the debugger and the inferior process involves the use of shared memory [Aral88a]. A large block of memory is allocated and shared between the debugger and the inferior process. Then the program to be debugged is dynamically linked and loaded into this shared memory. Control of the inferior process is then passed to the main procedure of this program. As a result, any update of data made by this program is readily observed by the debugger without any system call. Also, instructions of this program can easily be modified because they are loaded in the shared memory, to which the debugger has direct write access.

Chapter 6

A Toolkit for Building Debuggers

The previous chapter described the complexity and difficulties of debugger implementations. Implementations of debuggers involve at one end very low-level, operating system and architecture specific functions such as process control and decoding of machine instructions, and at the other end very high-level, sophisticated user interfaces. Unfortunately, the implementation of such tools is often done in an ad hoc fashion; only limited work has addressed designing generalized ways for building these tools [Fyfe91, Li91, Lint90]. Furthermore, the complexity of their implementations discourages experimentation with new debugging methodologies. Developers and researchers are often forced to build their tools by modifying existing debuggers, which may not be the ideal platform for their needs, or to build from scratch, which is too tedious and time-consuming.

This chapter addresses these problems and generalizes the basic functionalities required by all debuggers. It presents a layered model for designing and implementing debuggers, and describes experiences in applying such a model to build simple prototypes. The interfaces between adjacent layers are well-defined so that different implementations of the same layer can be interchanged transparently to other layers. Such flexibility is very important for building experimental debuggers, as well as for maintaining existing debuggers. By hiding the differences of different systems in the lower layers and standardizing their interface to the upper layers, the same higher-level *front end* can run on any system without modification. Likewise, by replacing the upper layers with different front ends, different tools can share the same lower layers (e.g., the audit trails analyzer described in Section 3.3.3 was built by simply programming Dalek code). Dividing functionality into layers is actually an application of good software engineering principles, which has not been followed before in this context. Our model can provide a generalized, system independent platform on top of which different debuggers can be built. Furthermore, our model is general enough that it is applicable not only to building debuggers but also to building monitoring tools in general.

We describe our layered model in Section 6.1, and its possible applications, such as building portable debuggers and debuggers for distributed programs running on heterogeneous systems, in Section 6.2. In Section 6.3, we outline our experience in building the working prototype of the layered model and simple tools that employ such a model. We found that using our layered model, the implementation of such tools, such as program tracers and profilers, is much simpler. Lastly, in Section 6.4, we discuss the importance of the layered model and how it differs from related work.

6.1. The Layered Model

Figure 6-1 shows the layers in our model. It includes the control process (i.e., the debugging or monitoring process), the inferior process, and the machine on which

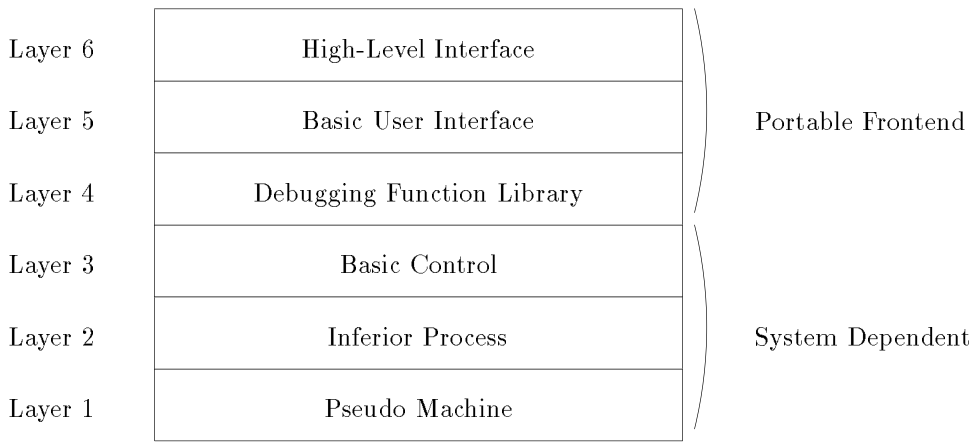

Figure 6-1. The layered model.

the inferior process executes. This model is divided into six layers. Each layer provides functions needed in implementing higher layers. Also, each layer represents a different level of abstraction in terms of the kinds of debugging operations and the representation of the inferior process's behavior. Implementation of each layer is independent of other layers and thus each layer can be replaced or modified transparently. As shown in the figure, the lower three layers implement functions that are system dependent, while the upper three layers are "portable". In other words, by providing a suitable implementation of the lower three layers for each pseudo machine and operating system, the same upper layers can be used on all these systems. On the other hand, the same lower layers can be used to build different debugging or monitoring tools. Each of these tools needs only to provide suitable upper layers.

Layer 1—Pseudo Machines

A pseudo machine is the overall environment on top of which an inferior process executes. This layer consists of everything that is not explicitly specified by the program's source code but is essential to its operation. For example, the pseudo machine corresponding to a UNIX process is the union of the physical machine and the operating system call interface. Whereas, say, the pseudo machine corresponding to a LISP program is the LISP interpreter. All pseudo machines should provide debugging support that allows a debugger to monitor the execution of an inferior process and to examine or modify its memory image. Also, the debugging functions supported by a pseudo machine should provide information about execution of the inferior process at the same level of abstraction as present in the pseudo machine.

Layer 2—Inferior Processes

An inferior process is the process that is being debugged or monitored. Each inferior process has its own address space. Pseudo machines with different architectures may create different problems in controlling the execution of an inferior process, depending on the kind of debugging support they provide.

Layer 3—Basic Control

This layer is the lowest layer of the control process. It interacts directly with the inferior process or the pseudo machine to provide basic functions that control the execution of the inferior process. These functions include initiation, termination, suspension, or resumption of execution, insertion of breakpoints, reading or writing of memory blocks or registers, and handling of interrupts and exceptions.

Layer 4—Debugging Function Library

This layer contains the machine independent control functions that are either pre-defined or can be added as needed. Together, layers 3 and 4 form a general-purpose interface to debug an inferior process. On top of this, different debugging *front ends* can be built.

Layer 5—Basic User Interface

Typically this layer includes a simple command interpreter that translates input from the user into specific requests for debugging actions. For example, in *gdb*, the command "`break <function-name>`" inserts a breakpoint at the beginning of the specified function. In *dbx*, the corresponding command is "`stop in <function-name>`", and in *sdb*, it is "`<function-name>: b`". Whatever the command syntax is, these commands are all translated by the corresponding command interpreter into, first, looking up the address of the specified function's entry point, and, second, inserting a breakpoint in that address.

Layer 6—High-Level Interface

Debugging a complicated program usually involves a large amount of interaction between the control and inferior processes. Specifying the proper actions of the debugger and the analysis of the information obtained can be a very tedious task and should be simplified. The high-level interface presents to the user the behavior of the program at a higher-level of abstraction (e.g., events [Bate88, Olss91b]).

6.1.1. Pseudo Machines (Layer 1)

The most important reason why the pseudo machines that execute inferior processes are included into the model is that programming languages are designed to execute on pseudo machines that do not resemble the physical machines. For example, LISP programs run on LISP machines whose basic operations are S-expressions [McCa65], and C programs run on machines that understand arithmetic expressions and statements. Thus, debugging functions supported by an operating system usually operate at a level of abstraction different from that described by a programming language, and a debugger must provide mechanisms to hide the differences and present to the upper layers an illusion that a pseudo machine actually exists. In Section 5.2.1, we described how we implement single-stepping at the program statement level in terms of basic operations such as breakpoints and single instruction stepping. Similarly, debuggers for other types of languages should also operate at the corresponding level of abstraction described by the language. Furthermore, implementations of these languages vary a lot: LISP programs usually run on top of an interpreter; C programs are usually translated to machine instructions; and others, such as Icon [Gris86, Gris90], are first translated to some intermediate language that is either interpreted or further translated to machine instructions. By including the pseudo machines as the lowest layer in our model, we can define high-level debugging operations in terms of the abstractions described by a programming language, and thus the model is independent of programming languages and their implementations. For example, we can define single-stepping as the execution of one basic operation provided by the underlying pseudo machine, and this might be the execution of a statement or expression in C, or an S-expression in LISP.

Another reason why debugging functions are defined in terms of pseudo machines' operations is that an inferior process might contain several different level of abstractions. It is desirable to use the same high-level debugging functions to work on any level of abstraction chosen by a programmer.

Consider, as an example, the implementation of an SR [Andr88] program. Conceptually, an SR program runs on top of an SR machine (Figure 6-2a). In the actual implementation, the SR compiler generates C code, which interacts with SR's runtime support (RTS). The RTS is an essential part of an SR machine, and it is also implemented in C. So, the same SR program can be viewed as a C program, which contains the RTS, running on top of a C machine (Figure 6-2b). The C program is further compiled into machine instructions that run on the physical machine (Figure 6-2c). Note that the actual machine instructions executed by the physical machine contain those that implement the operating system services.

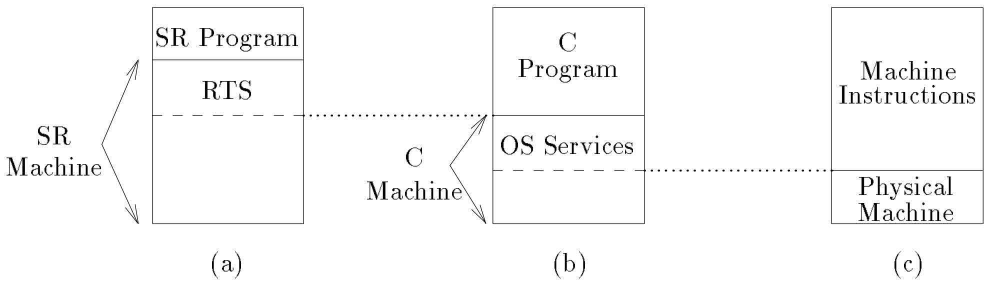

Figure 6-2. Different view of the same program.

Typically, programmers who debug an SR program need to stay only in SR's level of abstraction, and the debugger functions should work at the level of the SR machine. However, consider the case of debugging C programs. A C programmer debugging a C program might, under some situations, need to go down to the machine instruction level to examine, say, the stack or values of registers. For the same reason, some SR programmers (especially those who develop SR implementations) might want to examine the internals of the RTS or the C code generated by the SR compiler. They might to need to switch back and forth between the SR and C abstractions. For example, they might want to bring the execution of the program to a certain SR statement, and then switch to the C level to examine what corresponding C statements are generated, or what RTS services are being invoked.

With the debugging functions defined in terms of pseudo machine operations, a programmer can switch between different level of abstractions while using the same interface. That is, the same debugging commands, such as the setting of breakpoints or single-stepping, can be used regardless of the view of the inferior process. If the SR abstraction is chosen, breakpoints are set at SR statement boundaries and single step executes one SR statement. If later the C abstraction is chosen, breakpoints will be automatically set at C statement boundaries and single step will execute one C statement.

As we have seen in the previous chapter, the implementation of debuggers depends heavily on the operating system and hardware support. This dependency is also true in the pseudo machines. We require all pseudo machines to provide debugging support that allows a debugger to monitor and control the execution of the inferior process. Otherwise, many debugging functions cannot be implemented. For example, a LISP interpreter must provide to a debugger information such as the locations of variables and S-expressions, and must be able insert breakpoints at S-expression boundaries. The supporting functions that a pseudo machine should support are described in Section 6.1.3.

6.1.2. Inferior Processes (Layer 2)

It might seem non-intuitive to include inferior processes into the layered-model as they are the targets for debugging rather than part of a debugger. In a traditional approach for debugging, other than being controlled by the debugger, an inferior process executes in the same way as it would have been under normal condition and it plays no role in the debugging activities. After all, one should not expect an inferior process to debug itself.

However, advanced debugging techniques can make use of the inferior processes themselves to assist debugging. These techniques make resources already acquired by an inferior process available to the debugger as building blocks for new debugging functions. Some debuggers, such as *gdb*, allow invocations of functions defined in an inferior process to be part of an expression executed by the debuggers themselves. In other words, functions defined in the inferior process can be used to implement debugging operations. For example, an inferior process may already contain a function to print out the contents of a tree node in a certain format. A user can then define a debugging command using the debugger language to traverse the tree and use this function to print out content of the tree nodes.

Another way that an inferior process may take part in the debugging itself is to use the dynamic linking technique. As shown in Section 5.2.2, the address space of the inferior process can be used to dynamically link in debugging functions. Under the control of a debugger, an inferior process is loaded with debugging functions that are applicable and its execution can be redirected to these debugging function when necessary. In other words, an inferior process can be modified to debug itself—at least in many simple cases—and the debugger is reactivated only when more complicated debugging operations are required.

6.1.3. Basic Control (Layer 3)

This layer is the first layer implemented in the debugging process. It handles all direct interactions between the debugging and inferior processes. In terms of the implementation, this layer interfaces a debugger with the debugging functions supported by the underlying pseudo machines. Furthermore, this layer provides to the upper layers a standard set of functions to control inferior processes regardless of the structures of the pseudo machines on top of which they are executing. For programs that can be viewed at different level of abstractions, this layer provides a uniform interface so that the same set of functions can be used at all levels.

Many of the functions that should be provided by this layer have been described in Section 5.1.2. Here we describe all the necessary functions in greater detail and how they interface with the pseudo machines and upper layers. Pseudo machines should support most of the functions described below either directly or provide enough support so that they can be implemented efficiently.

(1) **Create**

This function creates a new inferior process or takes an existing process and puts it under the control of the debugger. A successful completion of this function should leave the specified inferior process in the "suspended" state. Control is then passed back to the debugger, which can then decide what to do next.

Pseudo machines that allow multiple inferior processes to execute in parallel should notify the debugger when an inferior process forks. The newly created process should be suspended and it should pass control to the debugger before execution begins. Similarly, when a multi-thread process creates a new thread, this new thread should also be suspended and the debugger should be notified.

All functions described below will have no effect on the specified inferior process unless it has been taken under the control of a debugger via this Create function.

(2) **Kill**

This function simply terminates the execution of the specified inferior process.

Note that both Create and Kill are actually interface functions to the corresponding routines that are supported and implemented by the underlying pseudo machine, which is responsible for any necessary initialization or clean up.

(3) **Suspend**

This function interrupts execution of the specified inferior process and then temporarily suspends its execution. This inferior process remains suspended until the Resume or Single-Step function is called. If the inferior process is currently suspended, this function has no effect.

(4) **Insert/Delete Breakpoint**

The breakpoint mechanism can be thought of as a synchronous version of the Suspend function. The Insert Breakpoint function marks a specified location in the code of an inferior process as a breakpoint. Whenever control is passed to this location, and before the code at this location is executed, execution of the inferior process is suspended and the corresponding debugging process is notified. In each location, at most one breakpoint can be inserted.[1] Calling the Insert Breakpoint function on a location where a breakpoint has already been inserted effectively does nothing.

Inserted breakpoints remain effective until they are removed by the Delete Breakpoint function or the corresponding inferior process terminates. In other words, breakpoints do not last beyond the lifetime of an inferior process.[2] Furthermore, they are not inherited by children of an inferior processes; i.e., a newly created child process of an inferior process should have no breakpoints inserted.

Conceptually, breakpoints are invisible markers in the code of an inferior process and their existence should not modify the code. However,

---

[1]More than one debugging operation can be associated with a breakpoint. The binding of breakpoints with debugging operations is implemented by upper layer functions.

[2]Functions implemented in the higher layer may record the breakpoints set in an inferior process and automatically load them back for subsequent runs of the same program.

some pseudo machines (in particular, most physical machines) support the breakpoint mechanism by replacing existing instructions with a special breakpoint instruction. Thus, an implementation of the Basic Control Layer on these pseudo machines should make sure that the insertion of these breakpoint instructions is transparent to the upper layers. In other words, when the upper layers need to examine the code of the inferior process, the original instructions rather than the breakpoint instructions should be shown. Furthermore, in the case of two inferior processes sharing the same code segment, an inserted breakpoint instruction is also shared by both processes. Since a breakpoint set in an inferior process should not affect any other inferior process, the pseudo machine and functions in this layer should make sure that an inserted breakpoint instruction is invisible to other inferior processes sharing the same code segment. Typically, this can be done by immediately resuming execution of other inferior processes that hit this breakpoint instruction.

(5) **Insert/Delete Watchpoint**

Similar to breakpoints, a watchpoint is another mechanism to suspend an inferior process synchronously. Unlike breakpoints, a watchpoint is triggered by a memory access to a specified location in the inferior process's address space, upon which execution of the inferior process is suspended. Watchpoints share most features of breakpoints: at most one watchpoint can be set at any location; they have to be explicitly deleted; they are not inherited by child processes; and their existence should not modify the code of the inferior process.[3]

There are three types of watchpoints. The first type is triggered when the specified memory location is read. The second type is triggered when the specified memory location is written. The third type is triggered when the specified memory location is accessed, i.e., either read or written. The Insert Watchpoint function provides a parameter for specifying which of the three ways should be triggered.

The implementation of the watchpoint mechanism depends heavily on the support of the underlying pseudo machine (see Section 5.1.2.5). Machines that do not provide direct support for watchpoints make their implementations very inefficient and difficult, if not impossible. On these machines, a watchpoint can be simulated by the Single-Step function. Execution of the inferior process is first suspended. Then, the code to be executed next is examined for any memory reference to the watchpoint. If this is not the case, the inferior process is single-stepped, and the above steps are repeated. Since single-stepping and examining the memory contents of the inferior process is expensive, simulation of watchpoints is usually much slower than a low-level implementation by the pseudo machines themselves.

[3]Similar to the implementation of breakpoints, more than one debugging operation can be associated with a watchpoint, and watchpoints set in a debugging session can be recorded and automatically loaded back in the subsequent sessions. But these features should be implemented by the upper layer functions.

(6) **Catch Signal**

The third mechanism to suspend an inferior process is to stop it whenever it receives a signal. A signal can be an interrupt delivered to an inferior process as a result of some external activities (e.g., expiration of an alarm clock) or it can be an exception generated internally by the processes's execution (e.g., division by zero). The Catch Signal function specifies which signal that is delivered to an inferior process should cause its execution to be suspended. Upon suspension, control is passed back to the debugger. If the debugger later wants the inferior process to see that caught signal, it should use the Deliver Signal function described below to explicitly deliver the signal.

(7) **Resume**

This function resumes execution of a suspended inferior process until: it hits another breakpoint, or watchpoint; receives a signal that is caught by the debugger; or the Suspend function is called again.

(8) **Single-Step**

This function resumes execution of the inferior process for one basic operation and then suspends it again. A basic operation is the simplest action that can be performed by a pseudo machine according to the semantics of the corresponding programming language. For example, on a physical machine, a basic operation is the execution of a single machine instruction. On a C machine, it is the evaluation of a simple expression or statement. On a LISP machine, it is the evaluation of a simple S-expression.

The Single-Step function can be thought of a special case of setting a breakpoint after the next operation followed by calling Resume. In practice, it is best implemented inside the underlying pseudo machine. Many physical machines, such as VAX [Digi86], Intel's 80386 [Inte86], and MIPS's R3000 [Kane89], provide hardware support for implementing the Single-Step function by automatically suspending the inferior process after executing one machine instruction. For those machines that do not have this kind of hardware support (e.g., SUN's SPARC [SunM87]), single-stepping through one machine instruction has to be simulated by inserting a breakpoint at the next instruction followed by resuming the inferior process's execution. If the instruction to be executed is a branch, then the debugger must calculate the destination address of the branch and insert the breakpoint accordingly. Such an operation is orders of magnitude slower than a hardware instruction-step. Pseudo machines should provide similar support according to their corresponding language semantics. Otherwise, the corresponding Single-Step function should be implemented in a way similar to the scheme described in Section 5.2.1.

(9) **Wait**

The Wait function suspends execution of the debugging process itself instead of the inferior process. The debugging process invoking this function is suspended until execution of the inferior process it is controlling is also suspended. This function is primarily used in conjunction with the Insert Breakpoint, Insert Watchpoint, and Catch Signal functions to synchronize execution with the inferior process. Typically, these functions are first called to specify when an inferior process should be suspended and its execution is resumed. The debugger then calls this Wait function to stop itself until the inferior process is suspended.

(10) **Deliver Signal**

This function sends the specified signal to the inferior process. Signals delivered via this function call are not caught by the Catch Signal function. If the inferior process is suspended at the time when Deliver Signal is called, the signal will be delivered it resumes execution or single-steps. Otherwise, the specified signal is delivered immediately. Hence, one way to implement the Suspend function is first to declare a signal to be caught and then send that signal to the inferior process by the Deliver Signal function.

(11) **Read/Write Memory**

These functions allow the entire memory address space of an inferior process to be examined and modified. These functions take as parameters a specification of a memory location to be read or written and a repetition count. In the case of a physical machine, these parameters would be the address and number of bytes to be accessed. In the case of pseudo machines for high-level languages, these parameters would be the name of variables or structures and, in the case of arrays, the number of elements to be accessed. Furthermore, these functions should be able to access variables outside of the current scope. Thus, these pseudo machines should provide support for mapping a variable name to the corresponding memory location. Note that for languages such as C, storage for local variables is dynamically allocated by the pseudo machine. Since the allocation method is language- and implemetation-dependent, it is the responsibility of the pseudo machine to map variables within or outside the current scope to the correct locations upon request. For example, most C machines allocate local storage in units of activation records on the stack (also know as stack frames), and so locating local variables outside the current scope might require traversing up or down the stack. However, if a particular C machine allocates storage for the activation records from the heap, a different searching algorithm should be used to locate the local variables.

(12) **Read/Write Status**

These functions allow examination and modification of information related to the execution status of an inferior process that is kept outside of its address space by the underlying pseudo machine. The kind of information maintained usually differs between pseudo machines. The only required information is the program counter or in general the position where execution will resume. Other information commonly maintained by pseudo machines is statistics on the resources acquired by the inferior process and status of the internal of the pseudo machines, such as the current state of the RTS of an SR machine.

An ideal pseudo machine should support all functions described above and implement them at the lowest possible level to allow maximum efficiency. A Basic Control Layer for such a machine is simply an interface that directly invokes corresponding support functions provided by the machine. Many interpreted languages, such as LISP, have interpreters implemented in software that closely resemble a pseudo machine. In these environments, most of the functions in the Basic Control Layer can be efficiently implemented in the interpreters themselves. However, the corresponding pseudo machines for some programming languages are just logical devices and might not exist at all. For example, compiled programs, such as those written in C, run directly on top of the operating system and physical machine. Debugging support from operating systems and physical machines is not only very limited, but is also aimed for debugging at the machine instruction level. That is, Single-Step always executes only one machine instruction, and Read Memory always requires a memory address. Under these situations, functions in the Basic Control Layer must perform a lot more processing to present to the upper layers an illusion that a pseudo machine actually exists.

Recall Figure 6-2, which shows how an SR program can be viewed at different level of abstractions. A debugger that supports debugging at multiple level of abstractions should contain a Basic Control Layer for each corresponding pseudo machine. When debugging at the machine instruction level, the pseudo machine is the physical machine. Thus, functions such as Insert Breakpoint or Single-Step should operate in terms of memory address and machine instructions. If a programmer wants to view the program at the C level, a Basic Control Layer for a C machine should be used and functions such as Single-Step or Read Memory should operate in terms of C statements or expressions and C variables respectively. The implementation for the Basic Control Layer for a C machine might be built on top of that for a physical machine. For example, the implementation of the C-level Single-Step function similar to what is described in Section 5.2.1 requires inserting breakpoints at all possible locations where control might be passed and then resuming the inferior process's execution. This function might be built in terms of the Insert/Delete Breakpoint and Resume functions at the machine instruction level. Likewise, when debugging at the SR level, a corresponding Basic Control Layer that works on an SR pseudo machine should be used, and it can be implemented on top of that for the C pseudo machine.

6.1.4. Debugging Function Library (Layer 4)

The lowest three layers (pseudo machines, inferior processes, and basic control) together provide a system-independent interface that allows a debugger to completely control the execution of an inferior process and manipulate its entire address space. A simple interface to the functions provided by the Basic Control Layer is sufficient to build a primitive debugger. In this section, we describe functions implemented on top of the Basic Control Layer that are essential for more sophisticated debugging.

On top of the Basic Control Layer in our model is the Debugging Function Library. As its name suggests, it is a collection of debugging functions—implemented in terms of the functions provided by the Basic Control Layer—that supports all the high-level debugging features available to a programmer. The next layer, the Basic User Interface, takes instructions from a programmer and translates them into invocations to the appropriate functions in this layer. Starting with this layer, and the rest of the layers on top of it, the exact functions that should be provided by each layers diversify because different debuggers might choose to have different debugging features. Some of the essential features, as well as those particular to the implementation of Dalek, are described in this section.

Functions provided in this layer can be invoked under two different mechanisms. Their execution can be initiated either from the user interface or from the inferior process. In the first case, the user interface accepts instructions either directly from a programmer or indirectly from a file and then translates them into invocations of functions in this layer. This is the case when a debugger is running in interactive mode accepting commands from the programmer. For example, when the inferior process is suspended, a programmer may issue a sequence of commands to examine the status of its execution or to insert or delete breakpoints. These commands are translated into invocations of functions in this layer. Control might eventually be passed down to the Basic Control Layer to actually carry out the debugging actions. In the second case, a programmer specifies the commands to be carried out when the inferior process stops executing as a result of hitting a breakpoint, watchpoint, or receiving a signal. When the inferior process stops, the underlying pseudo machine notifies the debugger, which then determines the cause of the suspension and invokes the associated debugging functions. Afterwards, control might be passed to the user interface to wait for the next command or it might be passed back to the inferior process by resuming its execution. For example, a conditional breakpoint is a breakpoint with an conditional expression attached. When an inferior process hits this breakpoints, the associated expression is evaluated (by functions in this layer). If the condition is satisfied, control is passed to the user interface. Otherwise, execution of the inferior process is resumed.

There are many ways to implement functions in this layer. One way is to identify all the required debugging functions and implement each of them separately. The user interface can then translate the user commands directly into invocation of the corresponding function. This approach allows simple and efficient implementation of the debugging functions, but the debugging features available are fixed and limited. Another way is to translate user commands into some form of intermediate code and build a general-purpose interpreter to execute them. All other debugging functions are then implemented in this intermediate code and functions in the Basic Control Layer are invoked directly from the interpreter when necessary. This approach is relatively less efficient because of the indirect execution through an interpreter. However, the kind of debugging functions is virtually unlimited and any new debugging function can be defined as needs arise.

In practice, a hybrid approach might be used. A library of commonly used debugging functions as well as the interpreter are provided by the debugger. Only special purpose debugging functions are interpreted; the rest are "hard-coded". Alternatively, dynamic linking can be used so that new, efficiently implemented debugging functions can be added on demand.

For debuggers that use a general-purpose programming language as a command language, the hybrid approach is desirable. Commonly used debugging functions such as conditional breakpoints or events should be provided by the debugger, while execution of any user-defined functions should be executed by the interpreter. This is also the approach employed by Dalek, except that the implementation of its interpreter does not follow our layered model. Dalek's implementation, which is based on *gdb*, is rather ad hoc. As a result, modifying or improving Dalek's functionalities is tedious and difficult. In fact, our experience with the implementation of Dalek was one of the motivations for developing the layered model.

Other than the command execution mechanism, another important mechanism in this layer is the initiation of debugging functions triggered by the suspension of the inferior process. This mechanism is similar to the implementation of an interrupt handler. It allows debugging functions to be bound with breakpoints, watchpoints, or caught signals. Whenever an inferior process is suspended, it checks the cause of suspension with the list of associated functions and then invokes the corresponding one.

In the case of Dalek, an additional mechanism implements the dataflow engine. This mechanism provides functions for defining and removing dependencies between nodes, delivery of data tokens, as well as other functions such as `event raise` and `event set-attribute` (see Chapter 3). Binding of operations with the corresponding events can share the same mechanism used to bind debugging functions with breakpoints.

6.1.5. Basic User Interface (Layer 5)

The major responsibility of the Basic User Interface is to accept instructions from a programmer and convert them into invocations of appropriate functions provided by the lower layers that actually implement the debugging operations. A simple debugger may discard the High-Level Interface and only support this layer. In this case, the debugger might not be very easy and convenient to use, but functionally it is a complete debugger because the actual debugging functions are all implemented in the lower layers.

The most important issue in designing a user interface is how well and how easily it allows programmers to express what they want the debugger to do for them. For a debugger, the quality of its user interface depends largely on the command language being used and the sophistication of the debugging functions provided. As described in the previous section, if the Debugging Function Library provides only a set of pre-defined debugging functions, the command language might simply be a list of commands each of which corresponds to a pre-defined debugging functions. On the other hand, if a general-purpose interpreter is provided, the command language can also be a general-purpose programming language. In this case, the expressiveness of the command language is greatly increased.

For those debuggers that uses a general-purpose programming language as command language (such as Dalek), it might be desirable to make the command language the same or closely resemble the language supported by the pseudo machine. If the language for writing a program and controlling the debugger is the same, a programmer does not need to spend extra time learning how to use the debugger. However, a different user interface has to be used for debugging programs written in different language. A good debugger should provide separate user interfaces for different languages and allow a programmer to specify which interface should be used.

A typical implementation of the Basic User Interface contains a main loop that repeatedly reads instructions from the input devices (e.g., keyboard, mouse, or file), invokes the parser to translate them into internal representation, which might be intermediate code for the interpreter or direct function calls to the predefined functions, and then passes control to the lower layers to actually carry out the debugging operations. The implementation of language parsers has been well-studied [Aho86, Mak91] and is not discussed here.

6.1.6. High-Level Interface (Layer 6)

A High-Level Interface contains all those functions that are not absolutely necessary for a debugger, but can help smooth the interaction between a programmer and a debugger and making the debugger easier to use. For example, a graphical interface may be used to specify the relationships between events in Dalek. Other kinds of assistance provided by the High-Level Interface vary from minimizing the number of keystrokes (by providing a command line editor such as [Fox89] or using a mouse to quickly select a command from a list of commonly used commands [Adam86]), using graphical symbols to present the change of data structures [Shim91], to the use of sound to indicate the behavior of an inferior process [Fran91].

6.2. Applications

Our layered model provides a framework for implementing debugging and monitoring tools. This model shares the advantages of other layered approaches for building complicated programs such as operating systems and network protocols. It simplifies the implementation by breaking down the program into more manageable modules, and associating with each a well-defined, specific functionality. More importantly, the isolation of different layers allows each layer to be individually implemented, verified, modified, or replaced without affecting other layers. As a result, researchers can experiment with different debugging and monitoring techniques by replacing only the layers involved, and developers can build different tools by putting in different implementations of higher layers while reusing the lower layers. For example, some UNIX systems allow processes to be accessed as files [Kill84] so that data can be transferred between the inferior and control processes much more efficiently than using the *ptrace* system call. If a debugger is implemented according to the layered model, it can be converted to use this new feature by simply replacing a few functions in the Basic Control Layer and the Pseudo Machine. The rest of the program can be safely left unchanged.

Tools that are built according to our layered model are more portable in general. All system dependent code is buried in the Basic Control Layer and below.

In other words, these lower layers provide a standardized, machine and operating system independent platform on top of which higher-level debugging functions can be built. Porting a debugger from one machine to another involves modification of part of these lower layers only. The front end of the debugger need not be changed. Furthermore, the concept of pseudo machines allows debuggers to be more easily ported to debug different programming languages. As long as the corresponding pseudo machine provides the necessary support for implementing a Basic Control Layer, debuggers for different programming languages can share most of the upper layer functions. In other words, the same user interface and higher-level functions can be put on tops of Basic Control Layers for different pseudo machines to debug different programming languages. The same concept also allows multiple views of the same program as discussed in Section 6.1.1. Debuggers for programs that go through several intermediate languages before finally being translated to machine instructions can include multiple Basic Control Layers, one for each pseudo machine corresponding to an intermediate language, while using the same upper layers. When switching from one pseudo machine to another, the corresponding Pseudo Machine and Basic Control Layer are used, and functions from the upper layers need not be changed.

The ability to implement debuggers that are portable to different systems is particularly important in distributed environments. Distributed programs are usually large and the same distributed program might contain processes running on machines with different architectures, operating systems, and even data representation formats. It is important that the same debugger can be used to debug every different process of a distributed program. Implementing a distributed debugger according to the layered model can eliminate the incompatibility of different operating systems and machine structures and allow a standardized user interface to be used. As described in Section 3.5.1, we propose that debuggers for distributed programs should be concurrent programs themselves, with a separate debugging agent attached to each separate process or thread of control. Each of these agents actually consists of the three lower layers, and the master debugging process is constructed from the upper three layers.

Our layered model can also be generalized for implementing other tools that require controlling the execution of other processes. Examples of such tools include execution profilers, virus monitors (tools that dynamically analyze if a given process is infected by computer virus), and other performance measurement or dynamic analysis tools. All these tools work on a *target* process and need to control its execution or access its memory image. That is, these tools share the same lower layers of a debugger, and thus can easily be built by replacing the front end (top layers) of an existing debugger. As a result, the layered approach facilitates sharing of code among a wide range of tools.

6.3. Experience

We have implemented the first three layers for C pseudo machines and physical machines. We have also built preliminary versions for debugging SR programs. The current implementation supports Sun 3 workstations (MC68020), SPARCstations (SPARC), and DECstations (MIPS R3000). We have also built several prototypical debuggers and program monitors on top of these layers.

The first version of these layers was developed on a Sun 3 workstation because most of the necessary debugging functions are supported by its operating system and hardware. This version was then ported to a SPARCstation. Since both machines run the same operating system (Sun Operating System Version 4), most of the code needs not be modified. However, the SPARC processor does not provide hardware support for single-stepping at the machine instruction level. As a result, single-stepping has to be specially handled by software (see description of the Single-Step function in Section 6.1.3). The port to the DECstation requires slight modification of the system calls because of the difference of operating systems (the DECstation that we use runs ULTRIX Version 4.1). The DECstation's MIPS processor has hardware support for single-stepping a machine instruction, but it is a heavily-pipelined machine that requires a delay slot after every branch instruction [Kane89]. The operating system treats the execution of any branch instruction and the instruction in its delay slot as an atomic action. Thus, single-stepping a branch instruction actually steps through two instructions, and inserting a breakpoint at a delay slot is not allowed. Nevertheless, we have not found any application that explicitly requires a breakpoint to be inserted at the delay slot.

With only three layers, our layered model has already shown its advantages. First, tools that we built on top of these layers can be executed on all the three supported systems without modification because the machine- and system-dependencies are all hidden by pseudo machines. Second, implementation of new tools is very fast and easy when compared to modifying existing tools or starting from scratch, because all the necessary low-level functions are provided by the layers and the interfaces to them are well-defined.

The tools that we have built include:

- an instruction counting program that reports the number of instructions actually executed by an inferior process. Basically, this is a very simple program

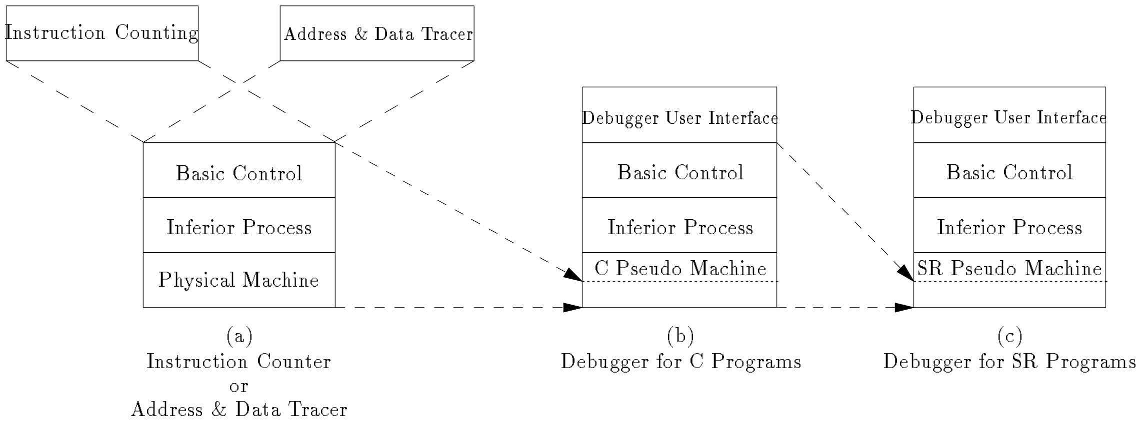

Figure 6-3. Applications of the layered model.

that repeatedly single-steps the inferior process until it finishes execution, and then outputs the number of instructions executed. Since this program works only at the machine instruction level, only those layers for a physical machine are used. Figure 6-3a shows the structure of this program. The lowest three layers for physical machines are used, and the instruction counting routine is built on top of these layers.

- a variety of address and data-tracing tools that monitor the data being transferred between the CPU and the main memory when an inferior process executes. During the execution of a process, machine instructions are being read into the CPU, and if an instruction involves memory access, content of the corresponding memory address will be read in or written into. Similar to the instruction counting program, these data tracers repeatedly single-steps the inferior process. In fact, they all share the same lower layer functions, as shown in Figure 6-3a. In addition to the single-stepping, these data tracers print out in each iteration the content of the current instruction and that of the memory address being accessed if that happens. The results can be used to analyze the patterns of data being transferred between the CPU and main memory, which is crucial to the overall performance of the CPU.

- a number of simple debuggers for programs written in C and SR. These debuggers use the layers for C pseudo machines. As shown in Figure 6-3, these layers are built on top of those layers for a physical machine and simulate a C machine. The additional functions required to complete the C pseudo machines are the symbol table handling routines and those for converting statement number to memory location (see Section 5.2.1). In other words, this C pseudo machine allows functions such as Single-Step and Insert Breakpoint to work at the statement level. Although these debuggers are very primitive (because we have not implemented the Debugging Function Libraries), they are working debuggers that allow programmers to control execution of inferior processes. The layered model also allow us to experiment with different approach in building debuggers for SR, which is a distributed programming language. As shown in 6-3c, the SR pseudo machine is built on top of the three lowest layers for C pseudo machines. In the current implementation, the SR pseudo machine is not complete because we have not put in the functions for mapping SR's statements to C's statements. Other than that, the experimental SR debuggers we built is powerful enough to control the execution of the inferior processes and examine the contents of variables.

The current implementation of Dalek does not follow our layered model, which was developed after Dalek was built. When we complete the development of the rest of the layers, we can improve our experimental debuggers and reimplement Dalek. The Debugging Function Library for Dalek should contain, among other necessary debugging functions, a general-purpose language interpreter and a dataflow engine for handling events. The User Interface Layer should contain a parser for Dalek's debugging language, and preferable a good command line editor. Other related tools, such as audit trails analyzer, can be built either by programming in Dalek code as described in Section 3.3.3, or more directly by replacing the parser in the User Interface Layer with one that accepts a special purpose language that is designed for that particular application.

An observation we made in implementing these lower three layers is that each operating system defines its own object file formats (e.g., a.out [Aout83], COFF [Girc88], etc.). These object file formats are all different and thus impose a lot of extra work in implementing the lower layers. On the other hand, these system dependencies further support the importance of the layered model in hiding the operating system-specific functions from the upper layers.

Although well-known in the UNIX community, the problem of too many object file formats does not seem to have a solution in the near future. In fact, instead of converging to a standardized object file format to be used by all systems, the number of different object file formats seems to be increasing. One attempt to solve this problem [Cham91] is to provide another layer that interfaces with the object files. This layer contains conversion routines for accessing object files. For each object file format, there is a corresponding set of conversion functions. Also, conversion functions for different object file format have the same interface.

6.4. Discussion

Despite the tremendous effort spent in building better debuggers and monitors, not much effort has been spent in studying the fundamental structure of the implementation of these tools. Apparently it is a common belief that the implementation of these tools is merely a programming exercise and does not merit much attention. However, as the complexity of such tools increases, the complexity of their implementations increases correspondingly. Furthermore, in the distributed environments, debugging and monitoring tools are not only complicated but may also run on heterogeneous systems. In such cases, a standardized model for implementation is even more important.

The idea of modularizing a complex program is not new. In fact, in describing their work, many people present the implementation of their debugging tools in terms of modules with different functionalities, such as graphical interface, execution control, event monitors, or communication manager [Aral88a, Bate88]. Splitting up the work of the debugger has also been proposed in [Lint90], which includes a proposal for a server called *Ndb*. Our approach differs from that work in that it provides a layered approach to the entire problem of debugging. We split up the functionality of even the lowest layers; i.e., the functionality of our lowest layers correspond roughly to that *Ndb* will provide. Our approach also splits up the higher layers of debugging. The *Blit* debugger [Carg85] employs an architecture similar to our concept of debugger agents. It contains two processes—one running on a host machine and another one running on a local workstation. High-level debugging functions are carried out in the host process while low-level debugging operations are carried out in the workstation. Similar to *Ndb*, it lacks the kind of fine-grain modularization in our layered approach that separates routines with different functionalities into different layers.

Fyfe [Fyfe91] proposes a toolkit called *psrv* for implementing debugging and monitoring tools. Similar to *Ndb*, *psrv* provides an interface for controlling and monitoring executions of inferior processes. Compared with our layered model, the functionalities of *psrv* is roughly equivalent to the lowest four layers combined together. As a result, it is less flexible than our layered approach in terms of portability, code sharing, and ease of modification.

Another toolkit for building debugger is *cdb* [Li91]. Unlike our layered model, *cdb* provides a number of "basic tools" as building blocks for debuggers. These tools do not have any hierarchical relationships with each other. Thus, porting them from one machine to another requires modifying all of them.

Our layered model can also be extended to parallel and distributed environments. As described in Section 3.5.1, there is a debugging agent for each individual thread in the inferior processes. Such debugging agents are actually composed of the lower three layers. No change is necessary for these three layers. However, functions that handle communication and coordination of the debugging agents have to be added to the Debugging Function Library. Furthermore, the High-Level Interface might need to implement multiple windows for displaying output and accepting input to each thread of the inferior processes.

The layered model is only one of the many possible ways to modularize and standardize the implementation of debugging or monitoring tools. The experience and feedback we gained from building the prototype debuggers and monitoring tools described above will allow us to refine this model and consolidate the exact interface between layers.

Chapter 7

Discussion and Future Work

This dissertation addresses various issues in software debugging. We introduced a new, event-based, dataflow approach for debugging sequential and concurrent programs. This new approach helps programmers obtain high-level abstractions of the execution of inferior processes. We proposed that debuggers should be controlled by a general-purpose programming language so that programmers can express flexibly and exactly what kind of debugging operations should be carried out. We demonstrated these ideas by implementing a prototypical debugger called Dalek for sequential programs. In terms of implementation techniques, we developed the concept of dynamic linking, which makes possible the implementations of a number of new debugging techniques, as well as many useful applications in other areas. We also built a working link/unlink editor that allows ordinary C programs to dynamically link or unlink compiled object modules. We described in detail the components of typical debuggers, the operating systems and hardware supports they require, and more importantly, a general layered model of how debuggers and related tools should be built so that they can be more portable and easily maintained, and similar components can be shared by different tools. We implemented a number of layers for different pseudo machines and built several experimental debugging and monitoring tools on top of them.

All in all, this dissertation attempts to present a more complete picture of solving the problem of software debugging: from methods and designs to implementations. However, there are still issues that we did not discuss. We study how we can make better tools to *assist* programmers to debug their programs, but our tools do not debug these programs *for* them. In other words, we do not discuss the problem of automatic or algorithmic debugging [Shap83]. As pointed out in the first chapter, software derivations, verifications, static analysis, and testings are important methods in assuring program correctness. Nevertheless, we focus our attention on studying debugging techniques because as long as human beings are involved, mistakes will be made, and debugging will be necessary. In fact, debugging techniques can be integrated with these other methods to provide even more powerful tools. For example, static analysis techniques are often limited by the combinatorial explosion of possible execution states and thus cannot be applied to complex programs. However, dynamic analysis on a portion of a program subject to a particular set of inputs might be possible if we combine a debugger with a static analyzer [Lo92, Olss91b]. Further study of how debugging techniques can be applied to these other methods should produce very interesting results. Finally, we chose to concentrate on more fundamental problems of debugging, such as how the execution behavior of an inferior process can be formulated in an easily comprehensible way, and not on how we can make debugging easier through the use of better user-interface, even though efficient communications between programmers and debuggers—either by means of graphics, sound, pointing devices, or other media—is essential. Furthermore, the debugging techniques we proposed do not have conflict with the implementation of any high-level user-interface because our layered model allows the development of user-interfaces to be independent of the rest of the debugger.

In the following sections, we discuss, for each of the new concept we proposed, the experience we have gained and the possible improvements that can be made in future. In particular, we address the dataflow approach, the debugging language, the concept of dynamic linking, the implementation of debuggers, and the layered model.

7.1. On the Dataflow Approach

Many existing debugging methods are event-based, with most of them being applied to debugging distributed programs. However, these methods are too passive and merely work as a filtering mechanism to eliminate irrelevant events. By integrating a fully programmable debugger language and the dataflow approach with event-based debugging, we turned these simple concepts into a novel method for debugging. Our approach allows a debugger to perform analysis on the behavior of the inferior process while it is executing, and actively respond with proper debugging operations to further track down the problem.

Dalek has allowed us to experiment and explore the potential applications of our dataflow approach for debugging sequential programs. For example, we have found out that our dataflow approach is excellent in correlating events that take place in different parts of a program and occur at different instance of its execution. Thus, this approach is very useful in tracking down errors that are resulted from a particular combination of function invocation sequence (e.g., the memory allocation/deallocation bug described in Section 3.2.5). This approach is also very useful in automating program tracing (see Section 3.2.2), which is otherwise very tedious and sometimes impossible to be done manually. The current implementation is not optimized and thus some of the debugging operations provided by Dalek is relatively slow. However, we believe that the efficiency of Dalek can be greatly improved by using the */proc* file system [Kill84] and other techniques described in Section 5.2. Other applications of the dataflow approach were discussed in Chapter 3. Being an entirely new approach for debugging, we believe that there are many more possible applications that have not been discovered, and more experience is required to fully explore its potential. One possible direction is to study how well this approach is applied to non-imperative languages such as Prolog, especially in debugging programs that involve backtracking.

We have described how our dataflow approach can be extended to distributed environments, and have built simple prototypes. However, we do not have a fully implemented distributed debugger that contains all the features we described. We believe our design would work, but there might be unforeseeable problems, which should be investigated first if this project is continued.

7.2. On Debugging Language

The current debugging language used in Dalek is intentionally low-level to ensure flexibility for experimentation. When more experience is obtained, the level of this language should be raised. This new debugging language should contain all the necessary language structures for general-purpose programming, and at the same time allows commonly used debugging actions to be specified easily. One possible solution is to use an ordinary general-purpose programming language and provide a standard library of frequently used debugging functions.

Another problem to be investigated is the semantics of the debugger language for debugging distributed programs. As illustrated by the example described in Section 3.5.3, semantics of some debugging operations have to be refined when used in distributed environments. When we have implemented a functionally more complete debugger for distributed programs, we can experiment and find out the exact details of how these debugging operations should work.

The current version of Dalek inherits from *gdb* a very ad hoc style of implementation, which has shown to be a hindrance for further improvements and modifications. Future experimentation should therefore be based on a complete re-implementation that follows our layered model. We have already implemented the lowest three layers for physical machines and C pseudo machines. What remains is to implement the Debugging Function Libraries and User-Interface layers.

7.3. On Implementation

Many of the implementation problems we describe in Chapter 5 directly result from constraints imposed by the operating systems or machine architectures. Some of the "techniques" to get around these constraints are actually not necessary if the operating system or machine architecture contained reasonable support for debugging. For example, on machines that do not provide hardware support for machine instruction-level single-stepping, a debugger must implement this function in software, which is usually orders of magnitude slower.

For better implementations of debuggers, more support from the operating systems and hardware is necessary. Summarizing our experience in implementing

Dalek and the functions in our layered model, an ideal environment for building debuggers should have:

- hardware support for single-stepping.
- hardware support for unlimited number of watchpoints.
- the ability to insert breakpoints that are truly transparent. That is, breakpoints can be inserted without modifying the inferior process's text segment. This can be done by allocating an extra bit for each instruction. When this bit is set, it means that a breakpoint is inserted before the corresponding instruction. This instruction will not be executed. Instead, the processor generates an interrupt to notify the operating system to stop the corresponding process and pass control to the debugger.
- a system service that allows a debugging process to include any section of an inferior process's address space into its own. That is, the operating system should allow a debugging process to have direct access to an inferior's address space. This can be done by modifying the virtual memory system so that portions of the virtual addresses of the debugging process and the inferior process are mapped to the same physical memory section.

7.4. On Dynamic Linking

The concept of dynamic linking and the *dld* package are more useful and have many more applications than we had expected. We originally designed a dynamic loader to allow debugging code to be loaded into an executing inferior process. But soon after we had decided to include the ability of unlinking an object module, we uncovered a large range of new applications for dynamic linking, which go beyond the area of debugging.

As stated in Section 4.1, dynamic linking combines the flexibility of modifying a program "on-the-fly" with the efficiency of executing native machine code. It therefore allows programming languages such as LISP and Prolog to be more efficiently implemented, and gives more flexibility and new functionalities to tools such as debuggers.

Dynamic linking also partially redefines the semantics of some programming languages such as C and Pascal. Programs written in these languages were considered static entities whose functionalities do not changed during execution. Dynamic linking changes these semantics by allowing functions to be modified at run time and even to be undefined after the beginning of execution. It would be an interesting topic to further study the impact of these added flexibilities to programming language design; either good or bad.

A slight modification of this idea of dynamic linking can twist further the semantics of many languages. Currently each process has only one symbol table and no symbol can have more than one definition. If we allow multiple symbol tables to be defined and provide functions for switching from one symbol table to another dynamically, a process can contain functions using the same name and the one defined in the currently "active" symbol table will be used when called. This feature can be used to implement function name overloading (as used in C++ [Stro86]) or simply to test different implementation of the same function without using two different names or creating two different executable files. Other applications, and the pros and cons of this features, remain to be investigated.

7.5. On the Layered Model

Introducing the layered model is a direct result of our experience in implementing experimental debuggers. We found that implementing a debugger from scratch in order to experiment with new high-level debugging approaches is both time-consuming and not cost-effective. For that reason, we implemented Dalek by modifying an existing debugger (i.e., *gdb*). However, the implementation of *gdb* is not flexible and systematic enough to allow easy modification. Furthermore, we observed that many tools, such as program monitors and profilers, require functions very similar to those in a debugger that handle the interactions between the debugging and inferior processes. These functions can be shared. Therefore, we decided to develop a better scheme for building these tools, which resulted in our layered model.

The concept of using a pseudo machine to hide the low-level details from the rest of the model is a novel one. There are simply too many different machine architectures, operating systems, and language implementation methods. Thus, any attempt to include all possible designs into a single model is bound to failure. By confining such differences into the lowest layer, all upper layers are free from any machine-, system-, or language-dependencies. Also, by defining a standard interface that all pseudo machines should provide (and requiring the debugger to simulate such a pseudo machine if the original one does not conform to this standard), the same upper layers can be used on any machines.

We have implemented the lowest three layers for C pseudo machines and described the other three upper layers. As pointed out in Section 6.1.4, the exact functions that should be provided in these upper layers diversifies as different debuggers might provide different debugging features. However, besides the few essential parts described in Section 6.1.4, there might be other functions that are commonly employed by many debuggers. Future studies should identify these functions and more concretely lay out the detail structure of these upper layers.

Bibliography

[Acce86] Accetta, M. J., "Mach: A New Kernel Foundation for Unix Development," *Proc. Summer Usenix*, pp. 93-112 (July 1986).

[ACM83] ACM, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging.* March 1983.

[ACM88] ACM, *ACM Workshop on Parallel and Distributed Debugging.* May 1988.

[ACM91] ACM, *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging.* May 1991.

[Adam86] Adams, E. and S. S. Muchnick, "Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations," *Software—Practice and Experience* Vol. **16**(7) pp. 653-669 (July 1986).

[Aho86] Aho, A. V., R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley Publishing Company (1986).

[Alle86] Allen, R., D. Baumgartner, K. Kennedy, and A. Porterfield, "Ptool: A Semiautomatic Parallel Programming Assistant," *Proc. of the Intl. Conf. on Parallel Processing*, pp. 164-170 (1986).

[Allm83] Allman, E., "SENDMAIL—An Internetwork Mail Router," in *Unix Programmer's Manual, 4.2 BSD*, University of California at Berkeley, Computer Science Division (August 1983).

[Andr88] Andrews, G. R., R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, "An Overview of the SR Language and Implementation," *ACM Transactions on Programming Languages and Systems* Vol. **10**(1)(January 1988).

[Aout83] A.out, "A.out(5)—Assembler and Link Editor Output Format," in *Unix Programmer's Manual, 4.2 BSD*, Vol. 1, University of California at Berkeley (August 1983).

[Aral88a] Aral, Z. and I. Gertner, "High-Level Debugging in Parasight," *ACM Workshop on Parallel and Distributed Debugging*, pp. 151-162 (May 1988).

[Aral88b] Aral, Z. and I. Gertner, "Non-Intrusive and Interactive Profiling in Parasight," *Proc. of the ACM/SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems*, pp. 21-30 (July 1988).

[ATT90] AT&T, *ANSI C & Programming Support Tools,* UNIX System V Release 4 Programmer's Guide, Prentice-Hall, Englewood Cliffs, NJ (1990).

[Baia86] Baiardi, F., N. De Francesco, and G. Vaglini, "Development of a Debugger for a Concurrent Language," *IEEE Transactions on Software Engineering* Vol. **SE-12**(4) pp. 547-553 (April 1986).

[Bate88] Bates, P. C., "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," *ACM Workshop on Parallel and Distributed Debugging*, pp. 11-22 (May 1988).

[Bate82] Bates, P. C. and J. C. Wileden, "EDL: A Basis for Distributed System Debugging Tools," *Proc. of the 15th Hawaii Intl. Conf. on System Sciences*, pp. 86-93 (1982).

[Bate83] Bates, P. C. and J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," *The Journal of Systems and Software* Vol. **3** pp. 255-264 (1983).

[Beck85] Beck, L. L., *System Software: An Introduction to System Programming,* Addison-Wesley Pub. Co., Reading, Mass. (1985).

[Blac88] Black, D. L., D. B. Golub, and K. Hauth, "The Mach Exception Handling Facility," *ACM Workshop on Parallel and Distributed Debugging,* pp. 45-56 (May 1988).

[Bore88] Borenstein, P., *THINK's Lightspeed Pascal User's Manual,* Symantec Corporation, Cupertino, CA (1988).

[Borl85] Borland International, Inc., *Turbo Pascal—Reference Manual,* Borland International, Scotts Valley, CA (1985).

[Brow88] Brown, M. H., "Exploring Algorithms Using Balsa-II," *IEEE Computer* Vol. **21**(5) pp. 14-36 (May 1988).

[Brue83] Bruegge, B. and P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism," *The Journal of Systems and Software* Vol. **3** pp. 256-276 (1983).

[Carg85] Cargill, T. A., "Implementation of the Blit Debugger," *Software—Practice and Experience* Vol. **15**(2) pp. 153-168 (February 1985).

[Carv86] Carver, R. H. and K. C. Tai, "Reproducible Testing of Concurrent Programs Based on Shared Variables," *Proc. of the 6th Intl. Conf. on Distributed Computing Systems,* pp. 428-433 (May 1986).

[Casw90] Caswell, D. and D. Black, "Implementing a Mach Debugger for Multithreaded Applications," *Proc. Winter Usenix,* pp. 25-39 (1990).

[Cham91] Chamberlain, S., *The Binary File Descriptor Library,* Free Software Foundation, Inc., Cambridge, MA (April 1991).

[Chan85] Chandy, K. M. and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems* Vol. **3**(1) pp. 63-75 (1985).

[Chas87] Chase, B. B., "Selective Interpretation as a Technique for Debugging Computationally Intensive Programs," *Proc. of the SIGPLAN 87*

Symposium on Interpreters and Interpretive Techniques, pp. 113-124 (July 87).

[Cher88] Cheriton, D., "The V Distributed System," *Comm. of ACM* Vol. **31**(3) pp. 314-333 (March 1988).

[Craw90] Crawford, R. H., "Topics in Behavior Modelling and Event-based Debugging," M.S. Thesis, Division of Computer Science, University of California at Davis (1990).

[Craw91] Crawford, R. H., W. Wilson Ho, R. A. Olsson, and C. E. Wee, "Extending the Dataflow Approach to the Debugging of Concurrent Programs," *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 222-224 (May 1991). Extended Abstract.

[Crow87] Crowe, M. K., "Dynamic Compilation in the Unix Environment," *Software—Practice and Experience* Vol. **17**(7) pp. 455-467 (July 1987).

[CSIM87] CSIM, "CSIM Reference Manual (Revision 11)," ACA-ST-252-87, Microelectronics and Computer Technology Corporation, Austin, TX (October 1987).

[Dale68] Daley, R. C. and J. B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS," *Communications of the ACM* Vol. **11**(5) pp. 306-312 (May 1968).

[DBX83] DBX, "DBX (1)," in *Unix Programmer's Manual, 4.2 BSD*, Vol. 1, University of California at Berkeley (August 1983).

[DEC88] DEC, "VMS Debugger Manual," AA-LA59A-TE, Digital Equipment Company, Maynard, MA (April 1988).

[DeMi79] DeMillo, R. A., R. J. Lipton, and A. J. Perlis, "Social Processes and Proofs of Theorems and Programs," *CACM* Vol. **22**(5) pp. 271-280 (May 1979).

[Digi86] Digital Equipment Corporation, *VAX Architecture Handbook*. 1986.

[Digi89] Digitalk Inc., *Smalltalk/V Mac: Tutorial and Programming Handbook.* 1989.

[Dubo82] Dubourdieu, D. J., "Implementation of Distributed Transactions," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 81-94 (1982).

[Elsh88] Elshoff, I. J. P., "A Distributed Debugger for Amoeba," *ACM Workshop on Parallel and Distributed Debugging*, pp. 1-10 (May 1988).

[Fetz88] Fetzer, J. H., "Program Verification: The Very Idea," *CACM* Vol. **31**(9) pp. 1048-1063 (September 1988).

[Fidg88] Fidge, C. J., "Partial Orders for Parallel Debugging," *ACM Workshop on Parallel and Distributed Debugging*, pp. 183-194 (May 1988).

[Fox89] Fox, B., *GNU Readline Library,* Free Software Foundation, Inc., Cambridge, MA (February 1989).

[Fran91] Francioni, J. M., L. Albright, and J. A. Jackson, "Debugging Parallel Programs Using Sound," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 60-68 (May 1991).

[Fyfe91] Fyfe, A., I. Soleimanipour, and D. Weatherford, "Design Issues for a Unix Program-Monitoring Toolkit," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 228-230 (May 1991). Extended Abstract.

[Garc85] Garcia, M. E. and W. Joseph Berman, "An Approach to Concurrent Systems Debugging," *Proc. 5th Intl. Conf. on Distributed Computing Systems*, pp. 507-514 (May 1985).

[Ging87] Gingell, R. A., M. Lee, X. T. Dang, and M. S. Weeks, "Shared Libraries in SunOS," *Proc. Summer Usenix*, pp. 131-145 (Summer 1987).

[Girc88] Gircys, G. R., *Understanding and Using COFF,* O'Reilly and Associates, Sebastopol, CA (1988).

[Gris86] Griswold, R. E. and M. T. Griswold, *The Implementation of the Icon Programming Language,* Princeton University Press, Princeton, NJ (1986).

[Gris90] Griswold, R. E. and M. T. Griswold, *The Icon Programming Language, 2nd Ed.,* Prentice-Hall, Inc., Englewood Cliffs, N.J. (1990).

[Hart85] Harter, P. K., Jr., D. M. Heimbigner, and R. King, "IDD: An Interactive Distributed Debugger," *Proc. of the 5th Intl. Conf. on Distributed Computing Systems,* pp. 498-506 (May 1985).

[Hebe90] Heberlein, L. T., G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber, "A Network Security Monitor," *Proc. 1990 Symposium on Research in Security and Privacy,* pp. 296-304 (May 1990).

[Hell78] Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," *SIAM Review* Vol. **20** pp. 740-777 (1978).

[Ho91a] Ho, W. Wilson., *Dld—A Dynamic Link/Unlink Editor,* Free Software Foundation (1991).

[Ho91b] Ho, W. Wilson and R. A. Olsson, "An Approach to Genuine Dynamic Linking," *Software—Practice and Experience* Vol. **21**(4) pp. 375-390 (April 1991).

[Hobb87] Hobbs, J., "Installed Shareable Images," *Dec. Professional* Vol. **6**(4) pp. 78-82 (April 1987).

[Hseu88] Hseush, W. and G. E. Kaiser, "Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language," *ACM Workshop on Parallel and Distributed Debugging,* pp. 236-247 (May 1988).

[Hume87] Hume, A., "Mk: a successor to make," *Proc. Summer Usenix,* pp. 445-457 (1987).

[Inte86] Intel Corporation Literature Department, *80386 Hardware Reference Manual,* Intel Corporation, Santa Clara, CA (1986).

[Joy86] Joy, W. and M. Horton, *An Introduction to Display Editing with Vi,* University of California at Berkeley, Computer Science Division (May 1986).

[Kane89] Kane, G., *MIPS RISC Architecture,* Prentice-Hall, Englewood Cliffs, NJ (1989).

[Kats79] Katseff, H. P., "Sdb: a symbolic debugger," in *Unix Programmer's Manual, 7th Edition*, Vol. 2C, Bell Laboratories (January 1979).

[Kauf88] Kaufer, S., R. Lopez, and S. Pratap, "Saber-C—An Interpreter-based Programming Environment for the C Language," *Proc. Summer Usenix*, pp. 161-171 (Summer 1988).

[Kepp91] Keppel, D., "A Portable Interface for On-The-Fly Instruction Space Modification," *Proc. of the 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 86-95 (April 1991).

[Kess90] Kessler, P. B., "Fast Breakpoints: Design and Implementation," *ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation* Vol. **25**(6) pp. 78-84 (June 1990).

[Kill84] Killian, T. J., "Processes as Files," *Proc. Summer Usenix,* pp. 203-207 (June 1984).

[Lamp78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. of ACM* Vol. **21**(7) pp. 558-565 (July 1978).

[LeBl85] LeBlanc, R. J. and A. D. Robbins, "Event-Driven Monitoring of Distributed Programs," *Proc. of the 5th Intl. Conf. on Distributed Computing Systems*, pp. 515-522 (May 1985).

[LeBl87] LeBlanc, T. J. and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers* Vol. **C-36**(4) pp. 471-482 (April 1987).

[Letw88] Letwin, G., "Dynamic Linking in OS/2," *Byte* Vol. **13**(4) pp. 273-280 (April 1988).

[Li91] Li, H. F., T. Radhakrishnan, and V. Krawczuk, "Cdb: A Toolkit for Debugging Distributed Programs," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 242-244 (May 1991). Extended Abstract.

[Lint90] Linton, M. A., "The Evolution of Dbx," *Proc. Summer Usenix*, pp. 211-220 (June 1990).

[Lo92] Lo, R. W., K. N. Levitt, and R. A. Olsson, "A New Approach to the Static Analysis of Array Bounds," *Technical Report*, Department of Computer Science, (January 1992).

[Loec84] Loeckx, J. and K. Sieber, *The Foundations of Program Verification,* pp. 113 and 218 John Wiley & Sons Ltd. (1984).

[Lopr89] Lopriore, L., "A User Interface Specification for a Program Debugging and Measuring Environment," *Software—Practice and Experience* Vol. **19**(5) pp. 437-460 (May 1989).

[Mak91] Mak, R., *Writing Compilers and Interpreters: An Applied Approach,* Wiley, New York (1991).

[McCa65] McCarthy, J. and S. Levin, *LISP 1.5 Programmer's Manual, 2nd Ed.,* MIT Press, Cambridge, MA (1965).

[McDo89] McDowell, C. E. and D. P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys* Vol. **21**(4) pp. 593-622 (December 1989).

[Mell89] Mellor-Crummey, J. M. and T. J. LeBlanc, "A Software Instruction Counter," *Proc. of the 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 78-86 (April 1989).

[Mill88] Miller, B. P. and J. D. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *Proc. of the SIGPLAN'88 Conference on Programming Language, Design and Implementation.*, pp. 135-144 (June 1988).

[Olss90] Olsson, R. A., R. H. Crawford, and W. Wilson Ho, "Dalek: a GNU, improved programmable debugger," *Proc. Summer Usenix,* pp. 221-231 (June 1990).

[Olss91a] Olsson, R. A., R. H. Crawford, and W. Wilson Ho, "A Dataflow Approach to Event-Based Debugging," *Software—Practice and Experience* Vol. **21**(2) pp. 209-229 (February 1991).

[Olss91b] Olsson, R. A., R. H. Crawford, W. Wilson Ho, and C. E. Wee, "Sequential Debugging at a High Level of Abstraction," *IEEE Software,* pp. 27-36 (May 1991).

[Pan91] Pan, D. Z. and M. A. Linton, "Supporting Multiple Execution Diagnostic Tools," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging,* pp. 254-256 (May 1991). Extended Abstract.

[Russ91a] Russell, M., *Ups—X11 and SunView Based Source Level C Debugger,* University of Kent, Canterbury (March 1991).

[Russ91b] Russell, M., *Porting the Ups Debugger,* University of Kent, Canterbury (November 1991).

[Saba90] Sabatella, M., "Issues in Shared Libraries Design," *Proc. Summer Usenix,* pp. 11-23 (June 1990).

[Shap83] Shapiro, E. Y., *Algorithmic Program Debugging,* ACM Distinguished Dissertations, MIT Press, Cambridge, MA (1983).

[Shim91] Shimomura, T. and S. Isoda, "Linked-List Visualization for Debugging," *IEEE Software,* pp. 44-51 (May 1991).

[Smit85] Smith, E. T., "A Debugger for Message-Based Processes," *Software—Practice and Experience* Vol. **15**(11) pp. 1073-1086 (November 1985).

[Soch88] Socha, D., M. L. Bailey, and D. Notkin, "Voyeur: Graphical Views of Parallel Programs," *ACM Workshop on Parallel and Distributed Debugging,* pp. 206-215 (May 1988).

[Spez86] Spezialetti, M. and P. Kearns, "Efficient Distributed Snapshots," *Proc. Sixth Intl. Conf. on Distributed Computing Systems,* pp. 382-388 (1986).

[Stal91a] Stallman, R. M., *Using and Porting GNU CC,* Free Software Foundation (January 1991).

[Stal91b] Stallman, R. M. and R. H. Pesch, *Using GDB: A Guide to the GNU Source-Level Debugger,* Free Software Foundation (July 1991).

[Stro86] Stroustrup, B., *The C++ Programming Language,* Addison-Wesley, Reading, MA (1986).

[SunM87] Sun Microsystems, Inc., *The SPARC Architecture Manual,* Part No: 800-1399-07 (August 1987).

[Tayl80] Taylor, R. N. and L. J. Osterweil, "Anomaly Detection in Concurrent Software by Static Dataflow Analysis," *IEEE Transactions on Software Engineering* Vol. **SE-6**(3) pp. 265-278 (May 1980).

[Tull83] Tulloch, J. and M. Alvarado, *Doctor Who—The Unfolding Text,* Macmillan Press (1983).

[Wile84] Wilensky, R., *LISPcraft,* W. W. Norton & Company, Inc., New York, NY (1984).

[Witt88] Wittie, L. D., "Debugging Distributed C Programs by Real Time Replay," *ACM Workshop on Parallel and Distributed Debugging,* pp. 57-67 (May 1988).

[Wood85] Wood, P. H. and S. G. Kochan, *UNIX System Security,* Hayden Book Co., Hasbrouck Heights, NJ (1985).

[Zern91] Zernik, D. and L. Rudolph, "Animating Work and Time for Debugging Parallel Programs Foundation and Experience," *Proc.*

ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 35-47 (May 1991).

Acknowledgments

I have been very fortunate to work with a group of people who are so supportive and encouraging throughout this project. Of whom the most important is my advisor Professor Ronald Olsson. His dedication to perfectionism has simulated all of those around him to satisfy with nothing but the highest quality of work. The proper attitude towards the search for truth and knowledge that he has demonstrated is no doubt the most valuable thing I have learned in the last five years.

I would like to thank Rick Crawford, Christopher Wee, and Carole McNamee for generating so many brilliant ideas during our meetings, and spending countless hours reviewing my earlier publications that have been incorporated into this dissertation.

I would also like to thank Professor Karl Levitt and Professor Mahadevan Ganapathi for their valuable comments on earlier drafts of this dissertation.

I can never say enough to thank Patricia, my dear wife, for her endless patience and encouragement throughout these years. Without her support, I would never have made it.

Finally, but most important of all, I must give thanks to my heavenly Father, who is the source of all my strength. His grace has always been sufficient for me.

Table of Contents

List of Figures