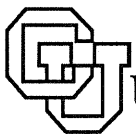
EXECUTION MONITORING FOR REUSABLE SOFTWARE COMPONENTS

Anthony M. Sloane

CU-CS-677-93 October 1993



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

EXECUTION MONITORING FOR REUSABLE SOFTWARE COMPONENTS

CU-CS-677-93 October 1993

Anthony M. Sloane

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

Abstract

This research addresses the provision of execution monitoring support for programs written using reusable software components. Execution monitoring encompasses the debugging and profiling of computer software. Components such as abstract data types, classes, or output from application generators are often used to reduce the effort needed to construct programs. Programmers can reuse the functionality of a component without needing to understand its internals. Source-level execution monitors are unsuitable for programs built this way because they insist on viewing the program at the programming language level. Programmers are often forced to understand the internals of a component in order to debug or profile their use of it.

We describe the design of a new execution monitoring framework for monitoring programs built from reusable software components. Information-hiding and abstraction, as previously applied to component functional interfaces, is extended to the development of their monitoring interfaces. We combine and extend elements from debugging and algorithm animation systems to allow monitors to interact with components without needing to know details of data representation or code structure. We demonstrate the applicability of the framework by designing monitors for a variety of components present in the Eli compiler construction system and the Icon programming language, as well as generic breakpoint and profiling monitors that can be applied to any component. These monitors have been built using a framework implementation called Noosa, whose design we also describe.

Thesis directed by Professor William M. Waite.

Acknowledgments

I am indebted to Bill Waite for giving me the opportunity to come to Boulder in the first place. Since that time he has been a reliable source of useful advice, encouragement and support, not to mention a most enjoyable collaborator. Most recently, he provided excellent comments on drafts of this thesis. Thanks also to past members of the Eli group: Bob, Steve, and Yogi, and current members Mark, Masayoshi, and Basim. Basim deserves special mention for valiantly reading a late draft of this thesis in short order and suggesting many improvements.

Thank you also to my committee: Ben Zorn, Dirk Grunwald, Wayne Citrin and Gary Nutt. They all contributed in significant ways to the work presented here. In particular, Ben deserves credit for his detailed reading of this thesis.

Malcolm Newey suggested Boulder for post-graduate work, so I have him to thank for the germination of this work. Nearing the end, Chris Johnson participated in useful discussions and provided valuable encouragement.

Special thanks to the many Boulder volleyball players who enabled me to escape from my work when that was needed.

Finally, infinite thanks to Nikki for supporting me in every way possible, most importantly, emotionally. In no small part this thesis exists because of her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many different software construction techniques can be collected under the single label *component-based software development*: Abstract data types support encapsulation of data and hiding of the routine implementations that operate on that data. Modular decomposition is a higher-level approach to designing systems so that subsystems can be developed and understood independently. Application generators (Cleaveland [1988]) produce components from high-level specifications. Object-oriented programming has been extensively applied to component-based program design and implementation.

Reusable software components are becoming an increasingly important part of software development. Where libraries of mathematical routines served the programmers of yesterday, graphical user interface components, components generated by application generators, and the like are serving today's programmers. Application complexity can be addressed through suitable reuse of existing components.

Collections of reusable components are usually designed to solve particular kinds of problems and are often intended to work together. We use the term *problem domain* (or just *domain*) to describe the kinds of problems addressed by a given set of components. We distinguish between three levels at which we can study a set of components. The *implementation level* is concerned with the code that implements components while the *component level* deals with individual component interfaces without regard to their implementation. The *domain level* considers a set of components as a coherent unit, a collection of cooperating interfaces.

Considerable previous research has been devoted to the study of software production using reusable components (Biggerstaff and Perlis [1989]). Topics studied include: the design of components maximizing reusability and evolution (Johnson and Foote [1988]; Meyer [1987]; Wirfs-Brock and Wilkerson [1989]), programming systems assisting the development of reusable components (Batory and O'Mallory [1992]; Neighbors [1989]; Neighbors [1984]), and the design and implementation of programming languages suitable for expressing reusability (Goldberg and Robson [1983]; Meyer [1992]).

This thesis addresses an aspect of component-based software production that has received little attention: *execution monitoring*. In particular, we address the implications of the use of reusable components on the debugging and profiling of software. The main

contributions of the thesis are:

1. *An extension of the central principles of component-based software production to include execution monitoring.* We show that it is advantageous for each component to be designed with monitoring in mind and describe the form that monitoring support should take: extra interface operations and event generation. Monitoring support should hide implementation details just as functional component interfaces do.

2. *A framework for component-based execution monitoring.* We describe a system organization that can be used at the domain level to enable interaction between program components, monitors[1] and programmers while maintaining levels of information hiding appropriate for component-based systems.

3. *Application of the framework.* We have built an implementation of the monitoring framework and used it to build new execution monitoring capabilities for two non-trivial domains: the Eli compiler construction system (Gray et al. [1992]), and string scanning in the Icon programming language (Griswold and Griswold [1983]). This work has shown that domain-level execution monitoring can provide sophisticated monitoring capabilities for a wide variety of components with only a small amount of work on the part of component builders. A number of general monitor design principles are also exposed.

1.1 Component-Based Software

Each component-based software construction technique contains a concept that represents a system-level building block: Abstract data types are combined (perhaps with some other code) to form programs, as are modular subsystems and classes in an object-oriented paradigm. Generators can operate with any of the other techniques; that is, they can generate data types, subsystems or objects depending on the situation. We refer to software building blocks as *components. Component-based software construction* refers to any process by which full systems are produced from constituent components.

When components are used to construct software they form a natural level at which to consider reuse. Each component *encapsulates* decisions about data representation and algorithms operating on those representations. Once a programmer has built a component to do some task, it can be reused by other programmers without regard to the details of the particular representations or algorithms used. Reuse applies whether a component is described by a piece of source code or a high-level specification.

The software construction process being used can make components available through libraries or similar means. The likelihood that a given component will be reused is a function of the task that the component performs (including the interface that it provides

[1]The term "monitor" as used here and in the rest of this thesis refers to "execution monitor" rather than "monitor" as used in the operating systems literature to mean either an operating system itself or a mechanism for mutual exclusion.

to other components) and the quality of its implementation. A goal of many software developers is to increase reusability. Few programmers today truly develop all of their programs from scratch; most use components of some sort even if only at the level of standard input/output routines. Component-based software can be expected to be more and more important in the future as application complexity and development costs increase.

This thesis applies to any component-based software technique. The relevant aspects of different techniques are described in Section 3.1. We restrict our attention to compiled sequential programs whose correctness is insensitive to timing issues. Interpreted environments permit approaches that are inappropriate for compiled programs. Many interesting research questions are raised by the monitoring of parallel programs or sequential programs with real-time constraints but we will not be addressing them here. Some of the methods we use have similarities to previously proposed solutions to monitoring these kinds of programs, and we shall note those similarities where appropriate. That is not to say that timing issues are irrelevant, because absolute execution times are important for profiling. The focus of this thesis is, however, on software whose functional behavior will not change due to timing differences introduced by monitoring.

1.2 Execution Monitoring

Execution monitoring (Plattner and Nievergelt [1981]) is a vital part of any software development process. Current technology does not permit us to construct complex software that is guaranteed to work the first time it is run. Even once a program performs its function correctly, it may not, say, perform it quickly enough. *Debugging* is a form of execution monitoring concerned with observing execution from the point of view of correctness; *profiling* considers execution with an eye on usage of resources such as processor time or memory space.

Component-based software construction places requirements on execution monitoring. To be true to a component-based technique, a system should support debugging and profiling at the component level. Unfortunately, programmers usually have to rely solely on source-level tools, thereby being forced to view their programs on possibly inappropriate levels. A central theme of this thesis is that tool and component developers should be attempting to provide higher-level support to complement source-level tools.

Consider a programmer debugging a component. He or she brings a certain amount of information along to a debugging session. There is information about the component in question: what it is supposed to do and how it is supposed to do it. There is also information about other components with which this one interacts. In accordance with the principle of encapsulation the programmer should be required to know *what* another component is supposed to do but not *how* it does it. We assume throughout this discussion that these other components operate correctly. Responsibility for ensuring their correctness lies with their developers, not the current programmer.

Suppose the programmer uses a component that implements generalized string storage

providing the following two operations:[2]

```
int store_string (string);
string lookup_string (int);
```

`Store_string` returns a unique integer for each unique string that it is presented, whereas `lookup_string` returns the string that corresponds to a given integer. Compilers, for example, routinely represent program identifiers in this way to reduce the overhead of storing and comparing them in other parts of the compiler.

This interface is sufficient to access the full functionality of the string storage. Strings can be stored and looked up as desired. However, more may be needed during execution monitoring. For example, suppose the programmer at some point wants to print the current contents of the storage. As is, the interface does not support such an operation so it is not possible to invoke it from a source-level debugger (as one could invoke `lookup_string`, for example). One alternative is to set tracepoints on each `store_string` call. This is not satisfactory because assembling the output from the tracepoints into a coherent view of the storage can be time-consuming and error-prone. Note also that it is not possible to use looping features in a debugger to call `lookup_string` repeatedly to get a picture of the entire storage. This is because the component rightly makes no guarantee about the range of integer values that will be used, allowing the use of any suitable technique such as hashing. Thus there is no general way to know which integers to pass to `lookup_string`.

In this case the programmer, who up till now has just been a client of the string storage component, is forced to break through the encapsulation of the component and examine its implementation. By looking at the code (assuming it is available) and the data of the component it may be possible to get the information needed. Of course, forcing clients to break encapsulations is not desirable. It places undue burden on the programmer to understand details that may only be incidentally related to the operation of their program. This burden is especially unwarranted when diagnosing a bug that involves the use of the string storage and not its operation *per se*.

Clearly the string storage component does not provide an adequate interface for monitoring. A natural reaction is to add an operation to print the current contents of the storage. In the case of a simple component like the string storage it is possible that a print operation could be designed that is acceptable to a majority of programmers. However, for more complex components this is less likely to be possible; consider the many possible layout strategies for tree-based data structures, for example. Thus we must provide a general facility that allows access to the data independent of the desired print format.

In Section 3.2 we investigate general access to abstractions supported by components. Component development normally concentrates on the run-time functionality required when designing interfaces. We argue that execution monitoring must also be taken into consideration and define the term *monitoring interface* to be the part of a component's interface specifically intended to support execution monitoring. The monitoring interface

[2]We use a C-like function prototype notation for program operation signatures. The return type is given first, followed by the name of the operation and any argument types listed in parentheses.

of a program is the union of the monitoring interfaces of all the components making up that program.

1.3 Events

Debugging relies on the ability to control the execution of a program. Execution is stopped at an appropriate point and data is examined to determine the state at that point. Normally, execution is controlled by facilities such as breakpoints.

With a source-level debugger, a program operation that expresses a desired stopping condition can be used to set a breakpoint by stopping when the operation is invoked (possibly also testing the operation's arguments). However, in many cases, suitable operations are not present. Current tools force programmers to examine internal details of components and describe breakpoints using predicates on data or line numbers in the code of operations.

Consider a component that encapsulates a general parsing scheme. It has a single operation that takes as input a context-free grammar and some text, parses the text and returns status information about the success of the parse.

During debugging a programmer may want to confirm that the grammar being passed in expresses the structure desired. The correspondence between parts of an input text and productions in a grammar is an integral part of the abstraction that the parser component supports. However, as it is, the interface of the parsing component does not support this part of the abstraction. A programmer wanting to examine the correspondences for a particular input text and a particular grammar is forced to violate the encapsulation of the parsing component. Depending on the complexity of the component implementation this task may be extremely difficult.

A better way to handle this case is to extend the interface of the parser component to fully support the abstraction. In general we can do this by introducing *events* that represent abstract points during the lifetime of the component; an event is *generated* when its point is reached at run-time. In the parsing case, an event that represents the association of a range of text with a particular grammar production is sufficient. Event instances are distinguished by their *type* and within each type by their *attributes*. For example, the parsing event might be of type *recognition* and have attributes representing the text range recognized and the grammar production used. Determining appropriate event types and attributes is thus part of the job of designing component interfaces for execution monitoring.

We describe the extension of component interfaces using events in Section 3.3. Events can be used to construct a powerful general breakpoint monitor that is independent of the actual event semantics. We discuss such a monitor in Section 5.10.

1.4 Profiling

Profiling of component-based systems also suffers from a reliance on source-level tools. These tools typically provide data about resource usage of source-level entities such as routines. A programmer may not have the necessary information to be able to map this data into component-level terms. For example, the routine that consumes the most processor time may be one that the programmer has never heard of. It could well be a routine hidden inside a component whose interface the programmer uses. We should not expect the programmer to profile their program at this level.

Our event-based approach allows profiling to be done at the component level. By generating events on entry to and exit from each component we can assign resources to components.

Breakpoint and profile monitors are two examples of *generic monitors*. They are characterized by the fact that they do not rely on the semantics of any part of a component's monitoring interface. We discuss generic monitors in Chapter 5. A profile monitor is discussed in Section 5.11.

1.5 Domain-Level Execution Monitoring

Components that have interfaces extended with extra operations and events can be used in a general framework for execution monitoring. We use the term *domain-level execution monitoring framework* to reflect that a collection of components can be regarded as collectively solving problems in a particular problem domain. For example, a collection of components implementing lexical, syntactic and semantic analysis operate in the compiler construction domain.

Our framework is summarized in Figure 1.1 and described in detail in Chapter 4. We identify the *subject* (the program being monitored), consisting of a collection of components, and the *frontend* consisting of the monitors. The programmer selects appropriate monitors and interacts with them to specify monitoring operations. The monitors in turn interact with the subject during execution to implement those operations.

Component operations can be invoked by monitors via message passing between the frontend and the subject. The results (if any) are returned directly to the requesting monitor. Monitors *register* interest in particular event types by specifying *event handlers*. Event handlers are invoked by the subject during execution when events are generated. Handlers can react in any way to events; sending messages back to monitors or stopping program execution are typical reactions.

1.6 Event Translation

Components are frequently implemented in terms of other components: Abstract data types often rely on each other. Generators sometimes produce specifications for other

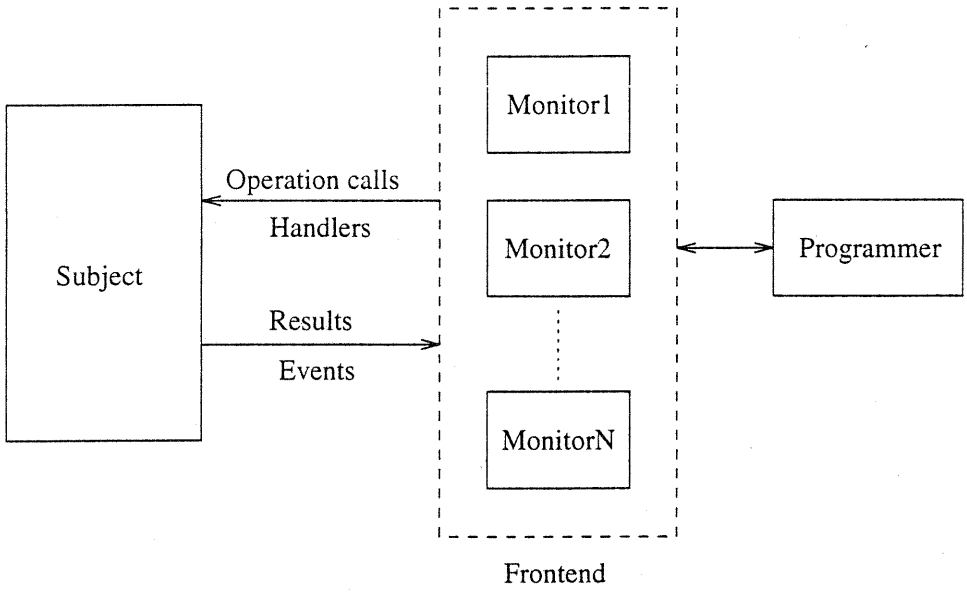


Figure 1.1: Domain-Level Execution Monitoring Framework.

generators. When inheritance is used in object-oriented programming, the class doing the inheriting is implemented in terms of the class from which it is inheriting.

Implicit in our discussion above on extending component interfaces to include operations for monitoring was the assumption that any dependence on other components is hidden. That is, a monitor need only call the relevant monitoring operation; the implementation of that operation may call a monitoring operation of another component if necessary, but this is hidden from the monitor and hence is also hidden from the programmer.

A similar situation exists when events are considered. When a component X utilizes another component Y it is necessary to translate events that Y generates into events appropriate for monitors of X. If this were not done, the reliance of X on Y would be revealed. The situation is somewhat complicated by the fact that Y may be used by more than one other component. Thus, X cannot simply translate all of Y's events.

In order to handle these situations we introduce the notion of an *event translator*. A component such as X is able to specify a translation to be applied to the event stream in order to hide dependences on other components. Note that it is crucial that X itself specify any such translation. Having some other part of the framework do it would violate the encapsulation of X. Event translation is described in Section 4.5.

1.7 Applications

The Eli compiler construction system (Gray et al. [1992]) makes extensive use of components to implement compilers. Libraries provide standard components that are used

in almost all compilers; tools generate other components from user specifications. Before the current work, monitoring an Eli-generated compiler was difficult because it required the programmer to understand many internal details of both the library and generated components. Chapter 5 describes a set of monitors that improve this situation considerably. We describe monitors for lexical analysis, syntactic analysis, string storage, error processing, and semantic analysis.

Many programming languages have simple-looking operations that implement complex processes on data. For example, Prolog's backtracking search algorithms are hidden behind a concise declarative notation. Often, language implementations do not provide debugging help for such operations. Also it is frequently hard to get any kind of performance data about the impact that the use of these operations is having on a particular program. Domain-level execution monitoring can be applied by regarding the implementations of complex operations as components. Section 5.8 illustrates this idea by describing a monitor for string scanning operations in the Icon programming language (Griswold and Griswold [1983]).

All of these monitors were constructed using an implementation of the *Noosa*[3] domain-level execution monitoring framework built for this thesis and described in Chapter 6. The implementation allows arbitrary C programs as subjects and utilizes an interpreted language to implement both event handlers and monitors.

1.8 Thesis Outline

In the next chapter, we consider previous research that has influenced our approach to the monitoring of component-based software. Chapter 3 defines what we mean by component-based software, considers why components typically do not adequately support execution monitoring, and describes extensions to interfaces that provide this support. Chapter 4 shows how components with extended interfaces can be used in a general framework to provide support for sophisticated execution monitors. Chapter 5 considers the development of monitoring interfaces from the perspective of the characteristics of the components being monitored and the requirements of monitors. The discussion is illustrated using monitors for the Eli compiler construction system, and string scanning in the Icon programming language. Chapter 5 also describes generic monitors: the class of monitors that do not rely on the semantics of monitoring interface operations or events. In Chapter 6 we briefly detail the interesting aspects of the framework and monitor implementations used in Chapter 5. We conclude with a summary of our results and a consideration of useful future work in Chapter 7.

[3]Noosa is a town on the east coast of Queensland, Australia noted for its fine beaches.

Chapter 2

Background and Related Work

The research described in this thesis has been influenced by many pieces of previous work. We begin this chapter by motivating the relevant related work via a consideration of the process of software reuse. The central aspect of software reuse that is important for our purposes is the relationship between the abstractions used by a programmer when writing a program and the abstractions supported by the monitoring system when monitoring that program. In Section 2.2, we observe that the source-level tools currently used by most programmers provide very poor support for reusable component abstractions of this kind. The rest of the chapter reviews a variety of approaches that other researchers have used to raise the level of abstraction that can be supported. Some of this work was directly targeted towards monitoring while some was aimed at other areas such as the construction of user interfaces or software visualization.

Theoretical techniques have been applied to the problem of execution monitoring but have not directly influenced our work. Some recent approaches are based on formal descriptions of programming languages; da Silva [1992], Berry [1991], Kishon, Hudak and Consel [1991], Kishon [1992], and Bertot [1991] describe some interesting work in this area. Unfortunately, since these approaches rely on semantic definitions of programming languages that are not usually available, they are not practical for application in most situations.

2.1 Software Reuse

This thesis addresses issues that arise when software is being reused. In a recent survey (Krueger [1992]), a variety of techniques for achieving reuse are described, ranging from high-level languages, through application generators, to software architectures. Each technique involves the application of the four processes of abstraction, selection, specialization and integration.

Abstraction forms the cornerstone of each approach by reducing the overhead of reusing artifacts. It removes the necessity for a programmer to completely understand an artifact before being able to use it. Given a collection of artifacts, a programmer constructing a program must be able to select which (if any) of them are going to be

useful. The process of selection is often keyed off classifications of the abstractions supported by the available artifacts. Classifications assist the programmer in finding out which artifacts are available and what their important characteristics are.

In many reuse approaches it is possible to have artifacts that provide general features while leaving some details unspecified. For example, a stack implementation can be provided without specifying the type of the objects that can be stored in the stack. When a programmer reuses the stack artifact, a particular type can be given to specialize the artifact for use in the current program. Often selection and specialization go hand in hand.

Once a set of artifacts have been selected and specialized they must be integrated with each other and any remaining artifacts developed from scratch. The result of integration is the constructed program. Different reuse methods apply different integration mechanisms. For example, a module interconnection language (Prieto-Diaz and Neighbors [1986]) may be used by the programmer to formally specify how different pieces of the program fit together. Alternatively, a reuse approach may call for artifacts to be automatically integrated based on a predetermined domain-specific structure.

The exact details of the selection, specialization and integration processes are clearly important aspects of any reuse technique. However, they are not that important from the point of view of execution monitoring. We are interested in the behavior of programs after they have been constructed. Thus we are concerned with the relationship between the abstractions that are employed by a programmer to construct a program and the facilities applied to the monitoring of that program. In this thesis we do not concentrate on one particular reuse technique; rather we use a model of software reuse that allows us to concentrate on the relevant aspects of all reuse techniques. In Section 3.1 we describe a process called *component-based software construction* that allows us to concentrate on this relationship. We assume that the input to the process is a specification of both the components that are to be reused and any application-specific ones. The process output is the constructed program. Thus selection is assumed to occur while the input specifications are being developed, while specialization and integration are encapsulated in the process itself.

2.2 Source-Level Tools

Our aim is to provide execution monitoring that is appropriate for component-based software. A useful starting point is tools intended to operate at the source level. By considering the application of these tools to component-based software and their limitations we gain insight into the requirements of a domain-level monitoring system.

2.2.1 Debugging

Programmers have been debugging programs ever since they started writing them. There is a wealth of literature on many aspects of debugging. Evans and Darley [1966] give an interesting early survey of debugging mechanisms. It is noteworthy that most of the

facilities present in source-level debuggers today are described in this paper. A notable exception is data breakpointing, which is only covered briefly in a discussion of using hardware traps to detect changes in register contents. Our purpose here is not to survey the entire debugging field. Good starting points for a more general consideration of current debugging technology are given by Johnson [1982] and Agrawal and Spafford [1989].

The GNU debugger GDB (Stallman and Pesch [1992]), and Dbx (Linton [1990]) (or the window-based Dbxtool (Adams and Muchnick [1986])), are typical examples of source-level debuggers in widespread use today. They provide excellent access to most aspects of program behavior in terms of program concepts such as variables, types, statements and routines. They incorporate large amounts of knowledge about the various source languages with which they operate. For example, expressions can be evaluated almost entirely in the debugger; only actual data values need be obtained from the program.

GDB and Dbx interact with the program being debugged through fairly narrow operating system interfaces. Parasight (Aral and Gertner [1988a]; Aral, Gertner and Schaffer [1989]) embodies a slightly different approach to source-level debugging that can increase the efficiency of many common operations. Parasight reduces the overhead of testing breakpoint conditions by loading monitoring routines called *scan-points* into the running program. Each scan-point can pause the program if its condition is true when its code location is reached. (A similar approach to breakpoints is described in Kessler [1990].) Parasight also has the program and the monitor run in the same address space (with suitable protection) and further reduces overhead by communicating using shared memory.

Because debuggers like GDB, Dbx and Parasight operate at source-code level, a central assumption of their design is that any part of the program that is to be debugged must be understood at the source level. For example, it is impossible to usefully examine the data of a piece of the program without knowing the names of the variables that hold the data. Similarly, it is hard to control the execution of the program without knowing the names of routines or the line numbers of relevant source locations. Under our assumptions of software reuse, variable names or source code may not be available to the programmer. Thus conventional source-level debuggers cannot be used directly to provide domain-level debugging.

Some researchers have extended source-level debugging tools to support more abstract modes of debugging. Duel (Golan and Hanson [1993]) is based on GDB. Although it provides very high-level data access using concepts such as generators, Duel does not really add any capabilities that help in the debugging of components because all of its operations must be applied directly to program variables.

2.2.2 Profiling

The most prominent early work involving profiling was a project analyzing FORTRAN programs (Knuth [1971]). Knuth presents a wealth of data about the programs and

their typical execution behavior. The primary conclusion of this study is the utility of a statement-level program profile for compiler design. Ingalls [1972] presents evidence from the same study to support the conclusion that profiles are of importance for typical programming tasks including debugging and testing.

Statement-level profiles are primarily of use when a programmer is working with a fairly localized piece of code. They can be especially useful when analyzing the performance of very high-level programs (Coutant, Griswold and Hanson [1983]). However, it is rare for modern-day programmers to examine a statement-level profile for their programs, the latter often being many thousands of lines long. A more global profile can be obtained if collections of statements are abstracted by routines. The gprof tool (Graham, Kessler and McKusick [1983]) is targeted at modular programs; that is, it assigns execution times for routines to the routines that call them. Using gprof, programmers are able to determine which modules (collections of routines) consume the most time, as well as the more traditional determination of individually expensive routines. Other tools such as qpt (Larus [1993]) provide similar information.

The Parasight system can be used for source-level profiling (Aral and Gertner [1988b]). It removes some of the overhead of profilers like gprof by having the profiler execute in parallel with the program being profiled. Scan-points are used to collect profiling information and communicate it back to the profiler. A distinguishing feature of Parasight is its ability to display profiles during the execution of the program, in contrast to other systems where profiles are only available once execution has completed.

A central assumption of source-level profiling tools is that the programmer is familiar with (or wants to be familiar with) the whole program at the source level. In component-based programs this is usually not the case. A profiling method based on routines will reveal more about the implementation of abstractions than may be desired by both the programmer and the designer of the abstraction. While routine-level profiles clearly have a place, there is a need for profiling at the component level.

2.3 Abstracting Program Execution

The abstraction of program execution is the key to an approach to execution monitoring for component-based programs. If we are to adequately support useful views of a program we must be able to suppress details that will complicate those views unnecessarily. In the rest of this chapter we describe various methods that have been proposed for abstracting the execution of programs above the source-level. Some systems apply more than one method; we group systems by our view of their most important contributions.

We mainly consider two aspects of the approaches that are important for component-based monitoring: *control abstraction* and *data abstraction.* Control abstraction refers to the support a particular technique gives to the abstraction of an executing program's locus of control away from the source code of the program. Data abstraction concerns the degree to which program data can be viewed without knowledge of the particular structures used by the program to represent that data.

Some work has been done to simplify debugging by applying highly structured abstractions to the programming process. Leavenworth [1977] restricts programs to those expressible in a particular domain-specific language. All control flow is abstracted by aggregate operations and side effects are not permitted. The debugging method involves traversing the data flow graph of the program analyzing each transformation of data. If a step is found that has correct inputs but incorrect output then a bug has been localized to that step. This method shows some overlap with our goals of monitoring at the domain level, albeit with some severe restrictions. It provides some evidence of the utility of providing monitoring capabilities at the level of the domain rather than of the source.

2.4 Events

Events have been extensively applied to the abstraction of program behavior. Schwartz argued the following in 1970:

> What we therefore need is a language whose fundamental entities are program events, and which allows us to search for events or combinations of events which we suspect to be anomalous and to display information concerning these events (Schwartz [1970,p.15]).

In this section we briefly survey previous approaches to monitoring sequential programs using events. Events have also been used extensively in parallel debugging and algorithm animation systems, which we survey in Sections 2.4.4 and 2.5.3, respectively.

2.4.1 Low-Level Approaches

One approach to event-based execution monitoring is to collect information about all operations that occur during execution and allow the programmer to query that information. Cohen and Carpenter [1977] describe one system that applies this approach at the source level using low-level events. Information recorded is keyed by the control flow through the program described as a trajectory of encountered program labels. Changes to variable values are recorded at appropriate places. An inquiry language enables *post facto* traversal of the trajectory using regular expressions. A similar approach is used in the Opium system (Ducassé and Emde [1991]). The obvious problem with this approach is the amount of time and space it takes to record this level of information. A better arrangement allows the current debugging tasks to guide the amount of information that is generated.

A dynamic approach to the use of low-level events for monitoring is used by the Dynascope system (Sosič [1992]). It achieves its power by using a low-level event stream that completely describes the execution of the program. Monitors can dynamically respond to the generation of events. Using low-level events allows them to be generated automatically. Programs execute under Dynascope as a mixture of machine code running at full speed and *hypothetical code* that is interpreted. Only hypothetical code can be monitored. Events are produced for each hypothetical instruction that is executed

by the interpreter; event attributes are the program and instruction counter, and the result address and value used by the instruction (if any). The abstraction capabilities of Dynascope are restricted to anything that can be inferred from the instruction event stream. Because hypothetical instructions are equivalent in level to machine instructions, inferring any component-level structure is difficult. Other systems that use machine-level events, such as AE (Larus [1990]), have similar problems.

2.4.2 Language-Based Approaches

Some programming language implementations have incorporated event-based models of debugging. Hanson [1978] describes the mechanisms used in an implementation of the SNOBOL4 programming language. It is possible to write sophisticated debugging systems for SNOBOL4 programs using these mechanisms. The available events are fixed and are based on source-level concepts: references to variables, statements, external interrupts, function entry and exit, or run-time errors. Unfortunately, reliance on fixed source-level events restricts the abstraction power of the approach.

Of further interest is Hanson's claim that "debugging aids (can be) written in the same language as the programs to be debugged." While this is mostly true, it is also the case that new built-in functions were added to support events so the debugging language is a superset of SNOBOL4. In this thesis we regard execution monitoring facilities as separate from the language with which they are used. While an overlap between the programming language and the debugging language is useful for source-level debugging tools, it is less important for domain-level debugging where no particular source language is implied—independence increases the scope of application of the monitoring tools.

The work by Tolmach and Appel in building a debugger for Standard ML (Tolmach and Appel [1990]) also applies language-level events. Their debugger adds instrumentation to functions being debugged to make them generate events corresponding to function application, value binding and so on. Again, inferring component-level behavior from these events is difficult.

The Loops Lisp system provides a paradigm called *access-oriented programming* (Stefik, Bobrow and Kahn [1988]) that is similar to Hanson's SNOBOL4 event mechanism. Actions can be attached to program variables using *active values*. When the value of a variable is changed by any part of the program the language implementation arranges for all of the variable's associated actions to be executed. Loops allows the action to execute either before or after a read or a write to the associated variable. Some recent work on program visualization makes use of mechanisms similar to active values (Haarslev and Möller [1990]). For example, they utilize slot *demons* to allow them to notice when slots of objects change. The Magpie programming environment (Delisle, Menicosy and Schwartz [1984]) also supports demons, this time for Pascal. Attached to identifiers, demons are anonymous procedures that are invoked when the identifier to which they are attached is accessed. The types of access that can be observed are restricted to variable reads and writes, plus routine entry and exit.

Using active values or demons it is possible to build monitors and programs independently. A monitor "attaches" itself to a program by appropriate reference to program identifiers; the program need not change at all when new monitors are added. Also it is not necessary to identify all places where variables may be accessed or modified in order to monitor those accesses or modifications. Unfortunately, this direct use of program identifiers makes monitors dependent on the source code of the program.

2.4.3 Higher-Level Events

Path expressions were first introduced to allow synchronizations between parallel processes to be described independently of the computations performed by those processes (Campbell and Habermann [1974]). *Generalized path-expressions* have been proposed for use in sequential debugging (Bruegge and Hibbard [1983]). They extend ordinary path expressions by allowing the primitive units of expressions, previously restricted to routine calls, to include any single-entry-single-exit piece of code in the program. Debugging is performed using *path rules* that contain a generalized path expression defining a program event and an action to be performed when that event occurs. Even though they allow more detail to be expressed than path expressions, generalized path expressions are still limited by their restriction to abstraction of the control flow of the program. Also, by naming code sequences in expressions, they force the user to rely on the source-level representation of the program.

A number of systems allow the use of higher-level events during debugging. The Dalek debugger (Olsson, Crawford and Ho [1991]; Olsson et al. [1991]), which is an extension of GDB, allows *primitive events* to be generated when breakpoints are reached. By defining appropriate primitive events and combining them into high-level events a programmer can abstract the behavior of a program away from its implementation details. However, since events are defined in terms of breakpoints, and breakpoints are defined in terms of source code concepts, Dalek still does not really provide control abstraction to the programmer.

Lazzerini and Lopriore [1989] and Lopriore [1989] also propose a model of program debugging based on higher-level events: "an event occurs when the evaluation of a conditional expressed in terms of program activity yields the value true. On the occurrence of an event, the actions connected with that event are performed by the debugging system." (Lazzerini and Lopriore [1989, p. 890]) As was the case with Dalek, the execution history information that is needed to express conditionals and actions is at the level of source concepts such as variables and source lines, so their primitives are not directly suitable for our purposes.

The Field programming environment (Reiss [1990]) makes extensive use of message passing between constituent tools for communication. One of the most heavily used message interfaces is that of the debugger, a conventional source-level debugger that implements access to the running program. Messages from the debugger about breakpoints are similar to the source-level events in the other systems we have discussed. Since Field is intended to be a source-level programming environment the rest of Field uses these

events without attempting to interpret them according to program abstractions.

Events are concerned with control abstraction, but are not really suitable for data abstraction in monitoring applications. Since events in most of these systems can have attributes attached to them it is theoretically possible to use an event stream to represent the complete data state of the program. All that is necessary is to generate an event for each change in any program data. The attributes of the event describe the change being performed, perhaps by identifying the data being changed and its new value. A monitor interested in program data can have a copy of the data and keep it consistent by watching the events that are generated. Clearly this is not desirable. Maintaining the copy is not only time-consuming, it wastes space through duplication, particularly if more than one monitor is interested in the same data. Also, in general it is impossible for a monitor to predict beforehand which pieces of data the programmer may want to examine. Thus there is no choice but to keep up-to-date copies of all possibly relevant data even if it is never looked at. Of course, for many realistic programs this is impractical, so we conclude that events do not by themselves provide an adequate basis for data abstraction in a monitoring system.

2.4.4 Parallel Execution Monitoring

An enormous amount of research as been applied to the problem of monitoring parallel programs. The interested reader is directed to McDowell and Helmbold [1989] and Utter and Pancake [1991]. In this section we briefly describe some projects that have affected the approach taken in this thesis.

Ponamgi, Hseush and Kaiser [1991] describe *data-path expressions*: path expressions with a concurrency operator (see also Hseush and Kaiser [1990]). Their MPD system is an extension of GDB that uses data-path expressions to express properties of the execution. Since their implementation is based on primitive events generated by GDB breakpoints, MPD is limited in the abstraction that it can support. An increase in expressive power is attained over that of plain path expressions because GDB breakpoints can include tests on program data.

The SPAE system (Grunwald et al. [1993]; Grunwald et al. [1991]), based on Larus' AE (Larus [1990]), uses some high-level events to enable different parallel execution architectures to be simulated. While not specifically aimed at monitoring, SPAE influenced the work in this thesis due to its reliance on abstract events. SPAE was primarily concerned with source-level operations that are easy to detect and low-level events such as memory accesses, so event generation could be inserted automatically at compile-time. Where high-level events are involved, another mechanism is required because it is impossible to automatically determine where events should be generated. Also, SPAE only incorporates a simple mechanism for simulators (the analog of monitors) to affect the execution of the program. For debugging we need much more support for execution control.

Behavioral abstraction is a debugging technique for parallel systems concerned with the modeling of program behavior and the comparison of actual behavior with the model

(Bates [1989]). Differences between the two can be used to identify problems. EDL (Bates and Wileden [1982]) and TSL (Rosenblum [1991]) are two specification languages that have been used to model system behavior. The main advantages of this approach over other event-based techniques for debugging parallel programs are that primitive events are not restricted to source-level concepts and abstraction is well supported by the ability to define new events as combinations of others. Sophisticated control abstraction can be achieved. In this thesis we observe that this kind of abstraction is not just useful for monitoring parallel programs. Also, we apply high-level control abstraction in conjunction with high-level data abstraction which is missing from these parallel debugging systems.

Liao and Cohen [1992] describe an application of behavioral abstraction techniques that reduces the runtime overhead of collecting monitoring data by having the programmer provide a formal specification of monitoring questions that are of interest, before the program is run. Their system generates an instrumented version of the program that is able to answer those specific questions expressed using temporal relations between program events. Unfortunately, programmers are often unable to specify before a monitoring session exactly which information they will want to see. For example, information revealed during monitoring may imply that other previously unspecified information is of interest. Many programmers are unlikely to use a system built on this technique since it is necessary to generate a new program to retrieve this new information.

2.4.5 Databases and Monitoring

Some researchers have proposed the use of database technology to implement monitoring systems. In Foley and McMath [1986] all of the current state of a process control system is stored in a database. Triggers are used to let the monitoring system know when things change. Their system supports interactive visualization creation based on attaching visual representations to the data stored in the database. The OMEGA system (Powell and Linton [1983]) uses a relational database to store all static and dynamic program information. Debugging operations are expressed in terms of a query language accessing the database. Snodgrass [1988] describes a similar approach to the monitoring of computer systems. A general problem with a database-centered approach is that monitor notification using triggers can be hard to implement efficiently.

2.4.6 User Interfaces

User interface development systems incorporate some elements of execution abstraction. For example, the Chiron system (Keller et al. [1991]; Young, Taylor and Troup [1988]) allows *artists* to maintain visual representations of the state of a program's abstract data types. Each ADT operation is invoked via a dispatching routine that calls the real operation and then notifies each interested artist of the fact that the operation was called. A problem with this approach is that only ADTs can be monitored by artists. The program must be structured as a set of interacting ADTs which may force significant changes to many existing programs. More seriously, since the unit of notification is

an ADT operation, artists cannot find out about changes in state that occur *within* operations. For many pieces of software and many visualization applications, updates to visual representations need to occur at a finer granularity than that provided by conventionally-designed ADTs (Brown [1988]). In this thesis we put the burden on ADT (component) designers to provide suitably abstract access to the ADT at this finer level.

The Model-View-Controller paradigm applied to the user interface of Smalltalk-80 (Goldberg and Robson [1983]; Krasner and Pope [1988]) separates the application (or model) from its graphical interaction (views for output, controllers for input). A model notifies associated views of changes by sending a message saying which parts of the model have changed. Views update their displays in response to this message by querying the model for any necessary information. Controllers map user actions into operations on the underlying model. This paradigm has been used to implement algorithm animation (London and Duisberg [1985]), but the complexities of mapping execution abstraction into this framework have steered the developers towards more abstract techniques based on constraint languages.

2.5 Software Visualization

Software visualization has been defined as the production of "graphical views or illustrations of the entities and characteristics of computer programs and algorithms." (Stasko and Patterson [1992]) As such, software visualization systems have much in common with the monitoring systems considered in this thesis. In this section we briefly describe the influential aspects of three applications of visualization: data structure display, visualization in programming environments, and algorithm animation. We confine ourselves to visualizations of sequential systems because parallel program visualizations mostly just concern visual representations of parallel interactions without any distinctive approaches to abstraction. Myers [1988] presents a fairly recent survey of visualization systems for sequential programs, while McDowell and Helmbold [1989] discuss the use of graphics in concurrent debugging.

2.5.1 Data Structure Display

Visualizations of data structures have proven to be effective for helping programmers understand the operation of their programs. The Incense system (Myers [1983]; Myers [1980]) implements automatic data structure display for the basic types of the Mesa programming language. Users can browse data structures using default displays and define customized displays if the defaults are inadequate. A similar approach is applied to linked lists in the VIPS debugger (Isoda, Shimomura and Ono [1987]; Shimomura and Isoda [1991]). The main problem with this kind of data structure display is its implicit reliance on source code representations of data. The programmer must understand the data types used to store program information, so abstraction is not well supported.

The University of Washington Illustrating Compiler (Henry, Whaley and Forstall [1990]) goes a little further by trying to infer the use of particular abstract data types such

as queues and graphs from their concrete representations in the source code. Standard illustrations are driven by the program during execution. While automatic visualization is an appealing goal, the current state of the technology does not permit many different data types to be inferred, so much code cannot be effectively automatically illustrated.

2.5.2 Visualization in Programming Environments

Programming environments are prime targets for the application of software visualization. By using visual representations of different aspects of programs under debugging and testing, environments can increase the ability of programmers to detect and diagnose problems. Ambler and Burnett [1989] present a good survey of all aspects of visual technology in environments.

Pecan (Reiss [1985]) makes heavy use of graphical techniques to display programs and their execution state. These displays are all designed around the fact that Pecan supports programming in Pascal, so they are all source-based. Pecan attains much of its power through its interpreted nature. Many of the visualizations would be much harder to do in a compiled environment.

A few projects have dealt with providing visualization for logic programming languages. The most prominent is the Transparent PROLOG machine (TPM) system that uses domain-specific displays of information to aid the comprehension of program execution (Eisenstadt and Brayshaw [1988]). Augmented AND/OR trees are used to show the execution of PROLOG programs from various views and TPM serves as a good illustration of how useful these techniques can be. The TPM visualizations are tied to a specific version of Prolog. Kusalik and Oster [1993] describe an architecture for visualization of logic programs that is independent of the particular language or system being used.

Each of these projects shows the utility of integrating visualizations with programming language systems. However, none of them really addresses the problem of providing visualizations that show the operation of a piece of code at an abstract level. Their aims were different: to show the source-level operation of code in a graphical way.

2.5.3 Algorithm Animation

Algorithm animation is the subclass of software visualization that has had the most influence on the work described in this thesis. Algorithm animation aims to provide visual representations of program executions that illustrate those executions in conceptually intuitive ways. Early work is reported by Baecker (Baecker [1975]; Baecker [1969]). The most prominent recent work in algorithm animation derives from Brown University. The BALSA I and BALSA II systems (Brown [1988]; Brown [1988]; Brown and Sedgewick [1985]; Brown and Sedgewick [1984]) provide convincing evidence that animations can help programmers understand the operation of algorithms. Another algorithm animation system originating from Brown University is TANGO (Stasko [1990a]; Stasko [1990b]). TANGO is based on a well-defined animation language that allows animations to be formally specified. Other recent algorithm animation work is described by London and

Duisberg [1985], Duisberg [1987], Delisle and Schwartz [1987], Helttula, Hyrskykari and Räihä [1989] and Bentley and Kernighan [1991].

The most successful application of algorithm animation to date has been to the teaching of algorithm design by abstracting the essentials of particular algorithms being studied (Brown [1988]). For example, an animation of a numeric sorting algorithm might display each number sorted as a rectangle of height proportional to the value of the number. Thus the ordering relation between the numbers is preserved in the animation while irrelevant details such as the actual values being sorted are suppressed. As numbers are exchanged during sorting, rectangles can be visually exchanged, reinforcing the process being performed by the program.

This kind of abstraction is similar to the kind that is needed when a programmer is monitoring a program containing components whose implementation is unknown. However, there is a crucial difference. The aim of an animation is to enhance the programmer's understanding of the operation of an algorithm. Visual abstractions enable it to be more easily grasped, but ultimately the information conveyed concerns the details of the code of the algorithm. In our setting we don't want to convey the details of the code; we want to convey the effect that the code of a component has on the abstractions implemented by that component. These levels of abstraction can be quite different. For example, the abstraction presented by a sorting component to a programmer normally does not include information about the particular sequence of item exchanges performed during a sort. It just specifies that the items come back sorted. However, graphical depiction of exchanges is central to most animations of sorting algorithms. A central idea of this thesis is that by raising the level of the interfaces between programs and monitors even higher than those used by animation systems we can support component-level abstraction.

The Zeus system (Brown [1992]) continues the BALSA line of algorithm animation systems. We briefly discuss some particulars of Zeus because it represents the state-of-the-art and applies methods typical of most animation systems. Zeus is notable in that it provides strongly-typed event interfaces, the ability to use parallelism in the implementation of an animation, and strong support for the use of color and sound (Brown and Hershberger [1992]). However, a number of features of Zeus make it unsuitable for execution monitoring. The main one is that algorithms to be animated must fit into a particular class hierarchy defined by Zeus and the animations are compiled with the algorithm. Since most algorithms are fairly short and can be rewritten from scratch this is not usually a problem in the application area targeted by Zeus. Unfortunately, it is not practical to require programmers to rewrite their programs completely in order to be able to (say) debug them. Most algorithm animation systems suffer from this problem.

Zeus' event specifications come close to providing adequate execution abstraction since events are high-level and can be generated at arbitrary times during execution of the algorithm or monitoring. However, there is potential for abuse of data abstraction because Zeus allows data to be declared as part of the event specification (Brown [1992] uses this approach for its example); it is then directly accessible to both the algorithm and the monitors. The approach followed in this thesis is to force all data access to go through a monitoring interface containing operations, thereby reducing the dependence

of monitors on specific data representations and increasing the reusability of monitors (Johnson and Foote [1988]).

A general problem with algorithm animation systems from a debugging standpoint is that they do not provide very good execution control. For example, Zeus only allows the algorithm to be run, asynchronously interrupted, or single-stepped one event at a time. For debugging purposes much more control is needed. It must be straightforward for the programmer to cause execution to stop at appropriate moments in order to examine the program's state.

Although not providing any better execution control, recent work has attempted to apply animation more directly to debugging (Mukherjea and Stasko [1993]). The Lens system aims to help programmers create animations on-the-fly during debugging. It appears that programmers familiar with the animation framework are able to construct simple animations for programs such as a bubble sort in a short amount of time. However, it is not clear that the benefit of having an animation outweighs the more significant effort needed to produce a useful visual representation for a more complex program. Indeed one can imagine that just designing the visual layout may consume more time than the average programmer in the middle of a debugging session would be willing to spend.

2.6 Summary

In their 1981 survey of execution monitoring (Plattner and Nievergelt [1981]), Plattner and Nievergelt observe that monitoring "can be specified in a satisfactory way only if the execution monitor is tailored to the programming language of the system." (p. 76) In this thesis, we generalize this requirement somewhat: execution monitoring must be tailored to whichever mechanisms the system provides to construct programs. In particular, if programs can be built from reusable components providing well-defined functional interfaces, the system must be designed to support those interfaces during monitoring. Just as interface abstraction is a property of programming languages strictly enforced by their compilers, it should also be strictly enforced by execution monitoring systems for programs written in those languages. The programmer should not be forced to deal with any details of the components that are not relevant for monitoring their program. While some of the systems described in this chapter could be used to approximate this kind of hiding, none of them support it as a system feature.

In many ways, this thesis unifies the approaches taken by debugger writers in trying to abstract execution with those taken by algorithm animation developers. Instead of source-level events, we use abstract events, similar to those used by animations and parallel debugging using behavioral abstraction but referring to component abstractions instead of algorithm or program implementations. Similarly, suitable data abstractions hide the source-level appearance of program data. In sum, there is no user access to source-level aspects of the components executing in the program.

Events are used as control abstractions that drive monitors. By designing events to match the abstractions used by the programmer to construct their program, and allowing execution control to be specified in terms of events, we attain a suitably high

level of execution control. Thus, like debugging systems and unlike animation systems, we elevate events to be user-visible elements of the monitoring system. Events also allow us to provide profiling on a component-by-component basis in contrast to existing profilers. Event translation allows components to hide the details of their implementations in terms of other components. While some previous systems allow events to be translated by the monitoring system, no systems that we are aware of permit the program itself to affect the event stream after primitive events have been generated.

Explicit in many of the approaches to monitoring described in this chapter is a dependence of the monitor on the internal data of the program. Thus, if the program evolves and changes to data representations are made, any monitors that accessed the old representation must be changed. Just as it is desirable to isolate programmers from the internal details of reused components, it is important to isolate monitors from those details. Thus, in this thesis all monitor interaction with the execution program is performed through well-defined interfaces that contain appropriate control and data abstractions. Monitors can be used with any program that supports the interfaces they require. By packaging monitoring support with the components themselves, that support can be reused just as the functionality of the components is reused.

To summarize the overall aims of this thesis, we turn to Stasko and Patterson's taxonomy for software visualization systems (Stasko and Patterson [1992]). In this paper they identify four dimensions that characterize software visualization systems: aspect, abstractness, animation, and automation. Aspect concerns the particular part of a software system that is being visualized. Abstractness (or amount of abstraction) concerns the level at which that aspect is presented to the user. A visualization is animated if it provides a continuous display of program operation as opposed to snapshots. Visualizations are automatic if they can be produced without any special work on the part of the programmer.

These dimensions are not only applicable to visualization systems. We can apply them to the goals of this thesis. The aspect of a software system we are interested in is its execution, including run-time state and control flow. We have a particular amount of abstraction in mind. Our monitoring systems must be able to support at least the amount of abstraction that is provided to the programmer by the functional interface of each component making up the software being monitored. Animations should be possible but are not the sole intended style of output. Full automation is essential, because the application area is execution monitoring for real software systems. A monitoring system is unlikely to be used if it requires extensive up-front work on the part of the programmer.

Chapter 3

Component-Based Execution Monitoring

This chapter begins with a description of the general software reuse model that will be assumed by the rest of the thesis: *component-based software construction.* In Section 3.2 we consider the problem of performing execution monitoring for software built in this way. In particular, we observe that the abstraction properties of component-based software must be preserved by any execution monitoring approach. Finally, Section 3.3 shows how each component can contribute to the preservation of abstraction by supplying a monitoring interface containing extra operations and event generation.

3.1 Component-Based Software Construction

Different software reuse technologies involve different methods of identifying components to be reused (Krueger [1992]). For our purposes the exact method used is not important. We assume that a programmer uses some procedure to arrive at a set of components that are to be reused in a program under construction.

The input to a component-based software construction process consists of 1) implementations of application-specific program components, and 2) specifications of the reusable components to be used in the program. The output of the process is the constructed executable program. For simplicity we assume that implementations of both application-specific components and reusable components are given in source code form written in a single programming language.[1] Thus the construction process will at a minimum contain steps to compile source code components and combine them into an executable.

In fact, traditional compilation environments for high-level languages are a simple form of a component-based software construction process. They accept source code components as input, compile them and combine them with any necessary library components (implementations of reusable components) to form the program. In this case the

[1]The examples given in this thesis use C (Kernighan and Ritchie [1978]) as the underlying source language.

necessary reusable components are referenced directly in the application-specific components so no separate specifications are needed. Any reference to a symbol of a library component will cause that component to be included in the constructed program.

A traditional compilation environment employs a simple form of reuse: library components referenced by symbols. More complex reuse technologies also fit our component-based software construction model. Consider a system that utilizes an application generator to achieve reuse. Any components that are not to be supplied by the generator are supplied by the programmer. The specifications of reusable components are used as input to the generator. They are translated by the generator into program components. All components are then combined to form the program.

We can also easily model construction processes that apply more than one reuse technique. For example, many generator-based systems also contain libraries of reusable components that do not need to be generated. A system could support the reuse of these components implicitly via reference to their symbols (as in a traditional compilation environment) or explicitly via a simple specification.

We can simplify the definition of our construction process somewhat if we consider the implementations of application-specific components to be specifications. Rather than requiring the system to produce a component implementation via some generation process or extraction from a library, they specify that the specification itself should be included as a component implementation. With this change we have the following definition:

Definition 1 *A* component-based software construction process *is a process with the following properties:*

1. *Its input is a set of user-supplied specifications.*

2. *Its output is an executable program.*

3. *Each input specification is interpreted as a request to include some set of reusable components in the output program.*

4. *The output program is a combination of the implementations of all requested components.*

It is appropriate to consider how this definition applies to reuse technologies that operate above the level of program components. For example, reusable software schemas allow higher-level artifacts such as algorithms to be reused (Krueger [1992]). A schema can specify an abstract form of an algorithm; it is instantiated to produce an implementation for use in a program. Thus the design of the algorithm is being reused rather than a specific implementation. Although our discussion leading to the definition above was based on reuse of source code components, higher-level reuse also fits the definition. In the case of schema-based reuse, input specifications describe both the schema to be reused and the instantiation to be created. The output program contains the implementation of the instantiation.

3.2 Execution Monitoring

Abstraction in software reuse implies information hiding or encapsulation. The details of implementations of reused components are hidden from the programmer behind abstract interfaces. Throughout program specification and design a programmer can rely on these interfaces to reduce complexity (Hoffman [1990]). The central theme of this thesis is that these encapsulations can and should be supported in other phases of development as well, in particular, when a programmer comes to monitor the execution of a constructed program.

Most computer systems provide a variety of execution monitoring tools. Some operate at the source-level; others lower. For most programmers the source-level tools are the most appropriate ones for everyday development. That is, since most programmers write programs in high-level languages, source-level monitoring tools are appropriate for these programmers.

Source-level monitoring tools aim to operate at the level supported by a high-level language. A compiler for such a language hides many details of the lower-level implementation such as register allocation or instruction selection, enabling the programmer to concentrate on machine-independent aspects of their program. Source-level debugging tools work similarly; they give access to program data and code in terms of the source code of the program, hiding lower-level details. Requests to display values of program variables are translated into appropriate accesses to machine registers or memory locations. Breakpoints at source locations are transparently implemented using patches to machine code at corresponding addresses. Source-level profilers provide similar hiding of low-level details. For example, a sampling profiler like gprof (Graham, Kessler and McKusick [1983]) performs a mapping from program counter values to source-level functions in order to estimate the amount of time spent in each function.

Generalizing from the source-level situation, we see that there is an essential parallel between the operation of a software construction process and the monitoring tools used on programs produced by that process. Any information that a construction process hides from the programmer should also be hidden when the programmer uses monitoring tools. For example, a source-level compiler transforms a high-level program into a machine-level implementation. Source-level monitoring tools hide the details of that transformation when allowing programmers to observe the execution of the machine-level implementation.

Consider the situation of component-based software construction. As discussed above, the construction process transforms specifications of the artifacts to be reused into program components implementing those artifacts. By analogy with the source-level case, tools that support the monitoring of component-based programs must hide all details of the transformations applied by the construction process. These tools operate at the *component level* as opposed to the source level.

We observe at this stage a parallel that can be drawn between source-level monitoring and component-level monitoring. At first glance it appears that a useful feature of any component-level monitoring tool would be the ability to monitor below the component

level, that is, at the implementation or source level. We can expect, however, that the need to do this will be confined to developers of the construction process or programmers writing component implementations designed for reuse. This state of affairs is analogous to the current situation with high-level languages: few programmers debug their programs in terms of assembly language. This capability is usually only used by compiler or debugger developers.[2] The presence of a component-level monitoring capability does not prevent the use of source-level tools. If source-level monitoring is needed for a component-based program, source-level tools can be applied to the component implementations produced by the construction process.

So what exactly do we mean by execution monitoring at the component level? Paraphrasing the discussion above, we mean that the view available to a programmer of an executing program must be at the same level as the view that they have of the program during design. This view is based on:

1. Detailed knowledge of the specifications being supplied to the construction process, and

2. Abstract knowledge of the components used by those specifications.

To illustrate this idea consider the development of a program using a reusable stack component. The programmer presumably has detailed knowledge of the use to which the stack is being put. That is, something in the application domain of the program is being implemented using the stack. In addition, the programmer has abstract knowledge about the stack component. He or she knows that a stack provides a storage repository into which items can be inserted and from which items can be retrieved. Also part of the programmer's abstract knowledge is the condition that an item can only be retrieved once for each time it is inserted and only once all items inserted after it have been retrieved.

Now consider the typical functional interface that a stack component would provide:

```
void push (itemtype);
itemtype pop ();
boolean empty ();
```

`Push` and `pop` allow items to be inserted and retrieved. `Empty` allows a client to detect when no more items can be retrieved.

While sufficient to perform stack operations, this interface does not fully support the programmer's abstraction of the stack component. For example, as mentioned above, one vital property of a stack is that items which are pushed on can later be retrieved. Hence during execution a stack contains some (possibly empty) set of items. However, there is no reasonable way for a programmer to determine this information while debugging using a source-level debugger. It is possible to put source-level breakpoints on the `push` and

[2]Debuggers such as the GNU debugger GDB (Stallman and Pesch [1992]) are not pure source-level debuggers because they support lower-level operations such as accessing register contents or disassembling code. Arguably, this is more a reflection of the target audience of these debuggers than a general need for lower-level facilities.

pop operations and combine the information from each activation of the breakpoint to get a picture of the state of the data structure. However, such an approach requires a large amount of work on the part of the programmer and is error-prone. Most programmers using source-level tools would instead consult the implementation of the stack, work out how the items are stored and get the information that way. Of course, this approach violates the abstraction.

Consider another typical debugging situation that results in a problem with the abstraction: the programmer runs the stack-based program and it signals an error because it attempts to retrieve an item from an empty stack. Suppose that the logic of the application implies that the stack shouldn't be empty at that point. The programmer now wants to determine when the stack became empty to get a clue as to where the problem lies. Using a source-level debugger it is probably possible to set a breakpoint that triggers at the end of the execution of the pop operation.[3] Each time this breakpoint is activated a check can be made for an empty stack. For our example this would probably be sufficient because there is only one operation that can remove items. If the component had more operations of this kind it would be much more tedious to use a source-level debugger in this way.

These two problems illustrate how component abstractions influence the facilities that monitoring tools must provide. In the first case, the fact that a stack maintains a set of items (the current contents) is a part of the abstraction of a stack. In the second we see that a component's abstraction may contain state information. A stack's state contains information about whether the stack is empty or not. It is useful to be able to access this state at run-time hence the existence of the `empty` operation. Our debugging example shows that it is also useful to be able to observe when a state change occurs.

3.3 Monitoring Interfaces

How do we allow monitoring tools to fully access the abstractions of components? As we saw in the previous section the functional interface to a component provides access to part of the abstraction supported by that component. Hence the natural way to proceed is to extend the interface of the component to give access to the rest of the abstraction. We use the term *monitoring interface* to refer to that part of a component's interface that is over and above that normally provided to access its functionality. The monitoring interface of a program is the union of the monitoring interfaces of the components from which it is constructed.

Not only is the use of monitoring interfaces a natural way to proceed, but under our software construction assumptions it is the only reasonable way. Component-based software construction puts heavy emphasis on the use of abstractions. The only parts of a monitoring system that have access to the implementations of abstractions are the components themselves. Giving some other part of the system access would violate the

[3] In some debuggers this requires setting a breakpoint at each exit point in the operation, thereby requiring abstraction to be broken since the source code must be accessed.

encapsulations of components and compromise their ability to evolve without affecting other parts. Chapter 4 will visit this issue again when we consider the overall structure of a component-based monitoring system. Building support for monitoring into components also has the advantage that the support becomes instantly reusable. That is, anyone using that component can take advantage of the support using whatever reuse technology is supported by the construction process.

3.3.1 Extra Operations

Operations in a monitoring interface are analogous to operations supplied in a functional interface: they provide access to the state of a component. In particular, they allow access to parts of the state that are not accessible via other operations.

Monitors need two types of access to component state: *query* and *update*. Query access makes it possible to display information to the programmer. Update access supports the implementation of debugging operations where the programmer wants to correct some incorrect behavior on the fly. This is analogous to a situation in source-level debugging when the programmer notices that a value has been computed incorrectly. He or she can correct the value using the debugger and continue debugging the rest of the execution.

In the context of the stack component of the previous section we can imagine some useful operations for monitoring. An operation that addresses the problem of the previous section is one that returns the set of items that are currently on the stack (in some predetermined order such as top to bottom). Monitors can use this operation to implement views of the stack contents during execution.

A useful update operation is one that gives monitors explicit access to the stack locations. For example, suppose that at some point during execution the programmer notices that one of the items on the stack is wrong. A monitor could allow the programmer to update the location containing the incorrect item to enable debugging to continue.

Operations added for monitoring purposes may implement an abstraction of a component different from that implemented by operations designed to be used to access the functionality of the component. The monitoring abstraction should include the functional abstraction; most likely it will be a superset. Sometimes query operations added for monitoring may be useful additions to the functional interface. For example, it is possible to imagine situations where other parts of a program may want to iterate over the items on a stack without having to retrieve the items and insert them again. Thus the monitoring operation that provides this ability may be generally useful even though it represents a slight weakening of the stack abstraction. Usually, update operations added for monitoring allow changes to the state of the component that are more incompatible with regular use. In the stack case we would be radically changing the nature of the stack component if we made available an operation that allowed updating of individual stack locations. As noted above, this operation is useful for monitoring but if included in the functional interface it would turn the stack component into an array component.

3.3.2 Events

Operations in a monitoring interface let a component provide access to its state at particular moments in time, i.e., the moments when monitors call the operations. Thus access is monitor-driven. In contrast, events are program-driven. They represent moments in time when a component changes state. Depending on the nature of a component's abstraction, state changes may have significance for execution monitoring. Events allow monitors to detect these changes without having to poll the state using component operations.

Events are *generated* by component implementations and are grouped by *event type*. An event type represents a particular abstract state change. The implementation of the component is constructed to generate events at appropriate places. More formally, an event of type T must be generated at all places where the implementation performs operations that correspond to the state change represented by T. Within a type, event instances are distinguished by *attributes*, values that are specified at event generation time. The attributes of T (if any) are used to specify which particular state change an instance of T represents.

Consider the stack component again. We had a problem determining where the stack became empty. We define an event type `emptied` that represents the state change when a stack has its last item retrieved. In the implementation of the component at all places where items are retrieved the component designer adds code to generate this event if the stack has just become empty. In our simple example there is only one such place: in the `pop` operation. Monitors can notice the event and react by (say) signalling an error and stopping the program. Chapter 4 describes the event handling mechanism in detail.

The `emptied` event type does not have any attributes because its instances do not need to be distinguished. Consider event types `pushed` and `popped` representing the insertion and retrieval of items, respectively. To distinguish their instances we need to know the identity of the pushed and popped items. Thus both of these events have a single attribute representing that item. A monitor could use these events to animate views of the stack during execution. We give examples of events and event attributes in more complicated settings in Chapter 5.

State changes are a kind of control flow. That is, events model progress through a component implementation. Control flow does not really play a big part in the abstraction of the stack component. Other components may have abstractions where control flow is more important. For example, consider a component that implements the lexical analysis portion of a compiler. An appropriate abstraction says that it will return a sequence of tokens.[4] Usually the component will be implemented with a loop reading characters from the input until a token is recognized. The existence of this loop is not part of the abstraction. Instead the programmer understands a stylized loop of the form:

```
while not end of file do
   return a token
end
```

[4]Most likely the structure of the tokens is given by a scanning specification supplied by the programmer.

where the body of the loop is implemented by the lexical component and the loop is present in a parser, for example. In this case the stream of events being generated by the lexer abstracts the stream of tokens being returned to the parser.

As noted in Chapter 2, in theory only events are needed to define a monitoring interface but this approach is not practical due to the time needed to transfer every piece of data to monitors, and the duplication of data between program and monitor, and between different monitors interested in the same data. Extra operations in the interface allow monitors to get to data when they need to and avoid duplication. A reasonable rule of thumb is the following: if data is maintained by the program then there is no reason for a monitor to keep a copy. On the other hand if there is some aspect of program history that is not maintained by the program but is needed by a monitor, then the monitor should store that data locally. Returning to the lexical analysis example above, if identifiers are stored in a table by the analyzer then a monitor can access them from there (given suitable operations). Actually, most compilers do not keep an explicit record of the token stream produced by the analyzer. Once a token has been used it is thrown away. Thus a monitor that wants to present the token history to the programmer must store it itself.

3.4 Summary

Monitoring interfaces support monitors just as functional interfaces support other components. Component implementation details are suitably hidden behind these interfaces. Within monitoring interfaces, data operations abstract component data and events abstract state changes.

Chapter 4

A Domain-Level Monitoring Framework

In this chapter we are concerned with the structure of monitoring systems targeted towards particular problem domains. We assume that a given domain has a collection of reusable components suitable for handling typical problems in the domain. Programs are built from some set of these components augmented with application-specific components. A monitoring system for a particular domain provides monitoring facilities for each of the available reusable components.

The general framework described in this chapter can be used to structure implementations of domain-level monitoring systems. The architecture of the framework emphasizes information hiding by utilizing monitoring interfaces to separate the program being monitored from the monitors. Flexibility is achieved by general mechanisms. None of the framework is tied to a particular problem domain because no semantic interpretations are placed on information. Later chapters will apply the framework. Chapter 5 shows how the mechanisms of the framework can be used to design powerful monitoring tools. Chapter 6 describes an implementation of the framework.

We begin with an overview of the architecture of the framework. Sections 4.2 and 4.3 describe the two main mechanisms supported by the architecture: events and operation invocation, respectively. Section 4.4 describes a method for allowing monitors to query the capabilities of the subject they are monitoring. In section 4.5 we point out a problem with using events in the presence of reusable components built using each other. Event translations are added to the framework to solve this problem. We conclude the chapter with a consideration of extensibility and a summary of the capabilities of the framework.

4.1 Architectural Overview

A domain-level monitoring system contains a collection of separate monitors for the different reusable components that are available in the domain of interest. This strategy mirrors the design of components as separate entities. When a component is reused, the monitor or monitors for that component are made available to the programmer. The

alternative is a monolithic structure that imposes unwanted overhead on a programmer who only uses some of the available reusable components.

Our monitoring framework architecture consists of two main processing units: the program being monitored or *subject* and the monitors themselves comprising the *monitoring frontend.*[1] At its heart the subject is the user's program. Thus it has an execution behavior that is determined by its function; some pattern of statement executions, data manipulations and control transfers comprise that execution. As discussed in Chapter 3, our goal is to allow the execution of components to be presented according to the abstractions that they support. Thus the subject must produce the results that it would normally produce (without monitoring) but the frontend must be able to interact with its constituent components in order to extract information for presentation to the programmer.

All interaction between users of the monitoring system and the subject takes place through the frontend. The frontend is a collection of monitor classes from which the user can choose. When a particular monitor class is selected an instance of that class is created and can be used to monitor the subject. More than one instance of a class can exist at any one time. Monitors can interact with any components that support the appropriate monitoring interface elements. In many systems there will be components such as abstract data types that are instantiated more than once. A browsing capability similar to that provided in Smalltalk-80 (Goldberg [1984]) would be suitable for creating monitors for particular instances of components. In the following we assume for simplicity that there is only one instance of each component.

The frontend is responsible for implementing all user actions. Some actions (like view scrolling or editing of program test input) can be performed entirely in the frontend. In the present discussion, we are not interested in the details of this kind of action. We concentrate on actions that require interaction with the subject, perhaps to determine the value of some program data or to set a breakpoint.

Necessarily, part of the frontend must be able to cause the program to execute, most likely multiple times during a single monitoring session. This functionality is the same as that required in a source-level debugging system, so we will not consider it further. Section 5.1 briefly describes a particular user interface for this part of a monitoring system.

During a monitoring session the program may be executed a number of times to examine different facets of its behavior perhaps using different test inputs. Monitors that have been created apply to each incarnation of the subject. To simplify discussion in the following, the term "subject" refers to any incarnation of the program under scrutiny.

[1] We deliberately avoid describing the subject and frontend as "processes". They may actually comprise one, two or many operating system processes in an implementation of the framework.

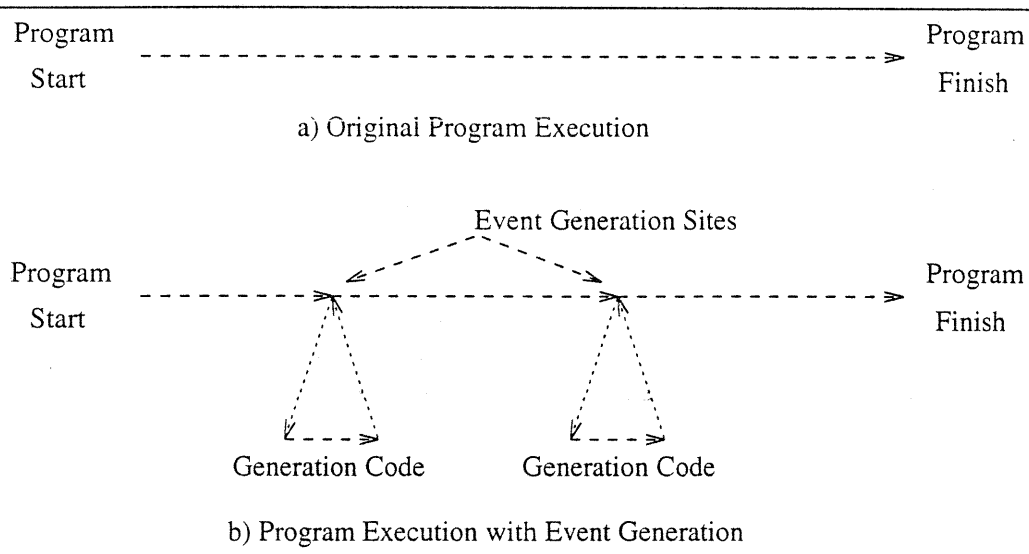


Figure 4.1: Program Execution: a) Original, b) With Event Generation.

4.2 The Event Mechanism

The subject and the frontend interact using two general mechanisms: events and operation invocations. The former are discussed in this section, the latter in the next.

Event instances are generated by the subject as it runs. They involve an interruption of its normal flow of control while monitors are notified of the occurrence of the event. Once any interested monitors have had a chance to react, normal execution of the subject continues. Figure 4.1 illustrates the differences that event generation make on the program execution. Each place where an event must be generated is called an *event generation site*. At each site, code is executed to produce the event. If the event being generated at a particular site has any attributes, they must be made available at the site and communicated to the event mechanism.

Current inferencing technology does not permit us to automatically determine event generation sites for complicated pieces of software. The placement of event generation code thus falls to someone who can manually examine the code for a component. We assume that the software construction process makes available some method for inserting event generation code into component implementations. Inline code and local or remote procedure calls are typical methods. Our aim is to encourage component developers to include event generation support when they are building components from scratch. In practice not every developer will do this, and there are many existing components without such support, so it will be necessary for third parties to do it. It is clear that whoever does it will need a fairly detailed knowledge of the workings of the component in question. The crucial point is that in the presence of component-based reuse there need only be one such person, in contrast to the current requirement that each programmer become familiar with the workings of a component in order to monitor their use of it.

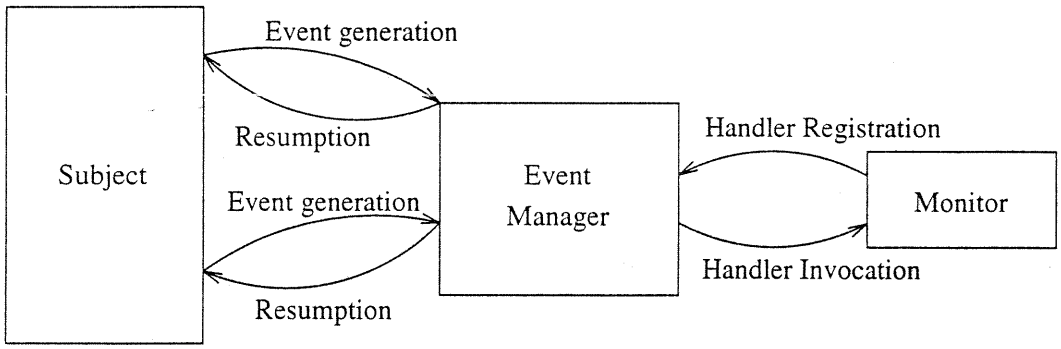


Figure 4.2: Control Flow Involving the Event Manager.

It is often useful for monitors to be able to generate events. We call them *synthetic events* because they are artificially created rather than being generated by the program. Synthetic events can be used as a form of communication between monitors. We describe how synthetic events can be produced in Section 4.3.

4.2.1 Event Management

The *event manager* is the portion of the monitoring framework that controls the dispatching of events to interested monitors (see Figure 4.2). When an event is generated the subject passes the event type and attribute information to the event manager. Conceptually, the subject blocks at this point, only to resume execution when the event manager returns control to it.

Monitors interact with the event manager in two circumstances: 1) when they want to register their interest in a particular event type (or cancel a previously registered interest), and 2) when the event manager wants to notify them that an event has occurred. Monitors are informed about the generation of events for which they have a current interest registered.

When a monitor registers interest in an event type, it specifies an *event handler* to be associated with that interest. An event handler specifies how a monitor would like to be notified when an event of an interesting type occurs. There may be more than one monitor that is interested in a particular event type. At event generation time the event manager invokes all of the handlers that are associated with the event type. Once all the handlers have been called, event generation is over and execution of the subject resumes. If there are no handlers for an event instance it is effectively ignored.

Event handlers can be canceled by monitors at any time. A canceled handler has no effect on any further execution. As an optimization it is useful for the event manager to provide a way for monitors to temporarily *disable* handlers and reenable them later without having to respecify the handler code. This allows monitors to *detach* themselves from the subject and be able to *attach* again with little cost. Detaching is useful when the

user of a monitoring system wants to use multiple instances of the same kind of monitor to get simultaneous views of the program state at various places during execution.

The power of an implementation of the execution monitoring framework derives largely from the power of the language used to express event handlers. At a minimum such a language must be able to express communication with monitors. For full flexibility, a general-purpose programming language with data storage and control structures is useful. The integration of the language with the framework must allow handlers (fragments of code on the order of small procedures) to be passed around and invoked when necessary. Chapter 6 describes the use of an interpreted language for this purpose.

4.2.2 Controlling Execution

Monitoring facilities such as animations can be implemented with the event mechanisms discussed so far. By defining appropriate handlers, monitors can update views of the program to produce an animation. Full-scale execution monitoring requires more control over the execution of the subject. In particular, to allow breakpoint capabilities to be implemented for debugging there must be some way for a monitor to specify that execution of the subject should *not* continue after an event has been generated.

Since the different monitors in a monitoring system act autonomously, we use the following mechanism for stopping execution: the return value from a handler determines the continuation action for the subject. When an event is generated, the event manager invokes all of the handlers for that event in some unspecified order. If any one of the handlers returns `TRUE`, execution of the subject should stop at that point. On the other hand, if all of them return `FALSE`, execution should continue.

Once the subject has been stopped, any monitor can get it going again by invoking a predefined operation in the subject's monitoring interface. We discuss this and other predefined operations in the next section.

If an event handler causes the subject to stop, it is safe to assume that the monitor to which the handler belongs knows that the subject has stopped. Other monitors may also want to know. It is clearly not practical nor desirable for each monitor to know about the handlers of all other monitors. To permit notification of stoppages we introduce a special synthetic event type called `stopped`. An event of this type is generated by the event manager whenever the subject is caused to stop for any reason. Monitors who need to know about stoppages can register handlers for the `stopped` event. Note that `stopped` events represent a global program state change rather than a state change in any single program component.

Many monitors provide views of current program data. If they are providing an animation they will update their display when appropriate data-related events are generated. On the other hand, animation may not be desired or may not be practical. In that case a monitor can use `stopped` events to allow it to update its view whenever the program stops. This practice is often a good compromise between continuous update and no update at all.

Usually when the subject stops there should be some feedback to the user about the

stoppage. Perhaps a breakpoint has been reached. Thus the `stopped` event has a single `locale` attribute describing which handler (or handlers) caused the stoppage. The name `locale` is used to indicate the parallel between this attribute and the location at which execution stops when a breakpoint is reached in a source-level debugger.

At this point we introduce the *Dapto* [2] language for describing monitoring interfaces by giving a description of the `stopped` event.[3]

```
event stopped "Execution of the subject has stopped"
   (str locale "Cause of stoppage");
```

In this example, we specify documentation for the event type and its attribute. The type of the attribute is given to be a string.

For convenience we also provide two other predefined event types: `init` and `finit`.

```
event init "Subject initialization" ();
event finit "Subject finalization" ();
```

Instances of init and finit are generated when the subject starts and finishes execution, respectively. All other event generations are guaranteed to occur between the init event and the finit event. Monitors can use the init event to initialize their displays for each run of the subject. Both of these events are useful for gaining control during debugging.

4.2.3 Time-stamps

Time profiling monitors need to be able to assign execution time to different portions of the subject. To make this possible all events must be time-stamped. It is sufficient for the event manager to attach the time-stamp to the event as an implicit event attribute. Event handlers can access this attribute as they would any other.

Use of time-stamps introduces a problem with correctness. In an implementation of the framework we can expect execution of the subject to take longer than it would with no monitoring present. Time-stamps must be adjusted to take into account the monitoring overhead otherwise information in profiles will be inaccurate. The event manager must subtract any execution time spent performing monitoring from the time-stamps on program events. We present some measurements of the actual perturbation in a framework implementation in Chapter 6.

4.2.4 Event Generation Algorithm

Figure 4.3 summarizes the event mechanism by giving the algorithm followed by the event manager when the subject generates an event. At each event generation site, control is transferred from the subject to the event manager, which executes this algorithm. Each

[2] Dapto is a town in central New South Wales, Australia not noted for anything in particular.

[3] Extensive use will be made of Dapto in Chapter 5. The full syntax of Dapto appears in Appendix A.

```
generate (event_type, attr1, attr2, ...)
   stop = false;
   locale = "";
   for each handler h for events of type event_type do
      ret = h (event_type, current_time,
               attr1, attr2, ...);
      stop = stop or ret;
      if ret then
         locale = concat (locale, name of h);
      end
   end
   if stop then
      generate (stopped, locale);
      wait;
   end
end
```

Figure 4.3: Event Manager Event Generation Algorithm.

handler is invoked with the event type, current time, and the attributes of the event. The name of each handler that returns `TRUE` is added to the variable `locale` which is used as the single attribute to the stopped event generated after all the handlers are done. The names of event handlers are specified by monitors at handler registration time. The `wait` statement terminates when the program is told to continue.

4.3 Operation Invocation

The event mechanism is subject-driven. In contrast, operation invocation is monitor-driven. Monitors use operations to access and possibly change the state of program components. Operations typically yield information that enables data views to be updated. Operation invocation can be regarded as a client-server relationship; the subject serves operation invocation requests from its monitor clients.

Operations must be executed by the subject. Since we are assuming that subjects are sequential programs this means that operations can only be performed when the subject is not busy doing something else. The only way the frontend can determine when the subject is not busy is through the event mechanism. When an event is being generated and handlers are being called the frontend knows that the subject is waiting until the event manager tells it to proceed. Similarly, if the program is stopped the monitors know that operations can be served. Thus the operation invocation mechanism must ensure that operation requests will be answered when handlers are being called and when the

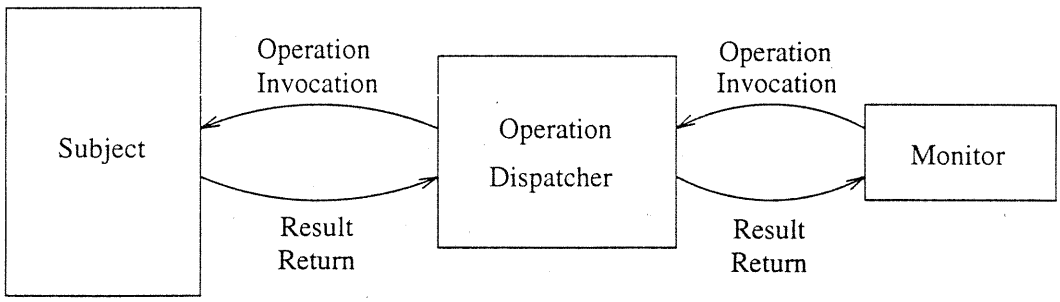


Figure 4.4: Control Flow Involving the Operation Dispatcher.

program is stopped.

The *operation dispatcher* is the portion of the monitoring framework that gets requests for operation invocations from monitors, dispatches them to the appropriate code in the subject and returns any results (see Figure 4.4).

In addition to the operations defined in component monitoring interfaces, we need two predefined operations. When the program is stopped we use a predefined `continue` operation to resume execution. Similarly, a `kill` operation allows us to cleanly get rid of the subject if the monitoring session is terminated while it is still running. The `continue` and `kill` operations are different from other operations in that they are not served by code in any component implementation. Here are the Dapto signatures for `continue` and `kill`:

```
operation continue "Continue from a stoppage" ();
operation kill "Terminate execution" ();
```

Recall that we want monitors to be able to generate synthetic events for inter-monitor communication. The easiest way to allow events to be generated by monitors is to make the event manager generation algorithm accessible via a data operation. For example, a monitor might invoke a `generate` operation with attributes indicating an event type and values for the attributes of that type. The event manager would then generate the event as usual.

4.4 Aspects

Most monitors rely on certain facilities being provided by monitoring interfaces of components in the subject. A monitor may provide a view of a certain component's data structure for example. We want to avoid a situation where the user of a monitoring system tries to use a monitor that is not applicable to the current subject.[4] While we

[4]Note that a similar situation does not arise for source-level monitoring tools because by definition all source-level programs must provide the set of facilities needed by the source-level tools.

could assume that the user knows which monitors are applicable a better solution is to allow monitors to query the subject to find out whether they are applicable. We define an *interface aspect* (or *aspect* for short) to be a part of the monitoring interface provided by a program. The monitoring interface of a reusable component defines an aspect of the overall monitoring interface of any program that includes that component. Support of an aspect by a program is a guarantee that all the operations and events of the corresponding component are supported by that program.

The software construction process can determine which aspects are supported by a program while it is collecting components with which to construct the program. Thus the result of the construction process should be the program along with its *monitoring database*. Somewhat analogous to the symbol table information generated by compilers for source-level debuggers, the monitoring database can be queried by monitors when they are presented with a program to be monitored. Only monitors whose required aspects are supported by the program will be available to the user for monitoring.

Generic monitors such as those described in Chapter 5 also make use of monitoring databases to obtain information about subjects. For example, event names and documentation strings can be used to provide domain-dependent information even in a domain-independent monitor.

4.5 Event Translations

We have seen how monitors interact with the components they are monitoring using events and operations. A monitor interacts through the event manager or operation dispatcher with the component or components that it is monitoring. Monitoring interfaces allow internal details of those components to be hidden. However, there is one situation where the mechanisms we have described so far are inadequate for complete information hiding.

Consider a monitor interested in a reusable component that is implemented in terms of another reusable component. Suppose further that the second component is of interest to some other set of monitors in its own right. Thus it is equipped with a monitoring interface that hides its internals. When the monitor invokes an operation of the original component, the implementation of that operation is able to make suitable calls on operations of the underlying component. All is well because the dependence on the underlying component is hidden from the monitor's view.

A problem is apparent when events are considered. Because the underlying component is of interest to other monitors it will most likely be generating events to be handled by those monitors. However, the existence of these events reveals the dependence of the original component on this component.

Consider an example of two components implementing two tables for storing strings, one allowing duplicates and the other not. The former component always returns a new index for each string inserted into the table. When a string is inserted into the latter table that is already present, the index of the existing entry is returned rather than the index of a new entry. In the spirit of code reuse, the developer of these tables may implement

```
gen = true;
for each translation t for events of type event_type do
   ret = t (event_type, current_time, attr1, attr2, ...);
   gen = gen and ret;
end
if not gen then
   return;
end
```

Figure 4.5: Event Manager Algorithm for Translating Events.

one in terms of the other. The table not allowing duplicates can easily be implemented using the other.

Now a user's program may use both of these components. However, the user should not be required to know anything about the implementation dependence of one on the other. Assuming that the user-level description of the components does not mention this dependence they should appear to be completely separate tables. However, both tables will generate events corresponding to insertions. A monitor for the table that allows duplicates will show insertions of strings into the other table, thus revealing the dependence.

Event translations are a mechanism designed to deal with this and similar situations. Just as operations can hide underlying implementation details, translations allow events to do so too. Translations are specified by components when a component that they use will be generating events that need to be suppressed or otherwise processed before they are passed to the frontend. This situation arises quite often in object-oriented systems where inheritance is used to reuse implementations.

The algorithm that the event manager follows is modified from that given previously to include the invocation of translations before any handlers are called. The return value of a translation is interpreted as an indication of whether this event instance should be generated or not. If any translation returns `FALSE` then the instance is not generated. The code sketched in Figure 4.5 is added to the beginning of the `generate` routine given in Figure 4.3. Each translation is invoked with the event type, current time and event attributes. Since we have no ordering relation between different translations, the semantics of having multiple translations for the same event type is only defined if those translations only ever affect different event instances. For full flexibility, the event manager must also provide facilities to allow translations to be dynamically disabled and enabled.

We would handle the dual table situation using translations as follows. Assume that the two components generate the following two event types for string insertions: A `string_stored` event is generated by the underlying string table when a string is inserted by either the user's program or the unique string table.

```
event string_stored
      "Storage of a new string in the string table"
   (int index "Index of the new string",
    str string "The string");
```

A `unique_string_stored` event is generated when a string is inserted into the unique string table.

```
event unique_string_stored
      "Storage of a string in the unique string table"
   (int index "Index of the string",
    str string "The string");
```

The unique string table component should specify the following translation. The `suppress` translation operates on `string_stored` events:

```
translation suppress string_stored
   "Throw away all string_stored events"
{
   return FALSE;
}
```

It should be initially disabled. `String_stored` events generated during the insertion of a string into the unique string table must be suppressed, so the translation should be enabled while such an insertion is taking place. This is appropriate because any such events reflect implementation details of the unique string table component and these details should not be visible elsewhere. Once the string has been inserted, the unique string table component generates a `unique_string_stored` event for the insertion and disables the translation again so that future `string_stored` events are generated correctly.

4.6 Extensibility

It is important to note that the provision of monitoring interfaces for use in the framework should not be restricted to reusable components. In many cases programmers will develop custom components for a given application. An implementation of the framework should make it possible for these custom implementations to make use of the event manager and the operation dispatcher. Extensibility is especially useful in conjunction with the generic monitors discussed in Chapter 5. Powerful monitoring can be provided for custom components with very little work on the part of the programmer. Programmers can also develop their own monitors as extensions of the base set of monitors provided by the system.

4.7 Summary

A domain-level monitoring system built according to the framework described in this chapter contains a subject, representing the executing instance of the program being monitored, and a frontend, representing the user interface. A frontend consists of a collection of monitors that can be applied to the monitoring of the subject. A monitoring database produced by the construction process allows monitors to tell whether they are applicable to the subject or not.

Each monitor interacts with program components through an event manager and an operation dispatcher. The former allows monitors to register interest in event types by specifying event handlers. Whenever events of interesting types are generated by the program, the registered handlers are invoked, enabling monitors to react to events. The operation dispatcher enables monitors to invoke program operations when the subject is stopped or is invoking an event handler.

Monitors can control the execution of the subject by causing it to stop when appropriate events are generated. Once execution is stopped, predefined operations allow it to be continued or terminated. Predefined events allow monitors to detect when the program has been started and when it finishes, as well as when it stops in the middle of execution. Event translations permit program components to hide implementation details by altering or suppressing event generation.

Chapter 5

Monitors and Monitoring Interfaces

The domain-level monitoring framework described in Chapter 4 is extremely general. It is important when evaluating such a general design to gain experience applying it to a variety of real-world situations. Only then can we obtain an accurate measure of its applicability.

This chapter presents results from the successful application of the framework to the monitoring of a number of different kinds of components. Sections 5.2 through 5.7 present monitors that provide domain-level monitoring for the Eli compiler construction system (Gray et al. [1992]). Eli is an extremely useful testbed for evaluating the monitoring framework because it contains a wide variety of reusable components ranging from relatively simple fixed components to complex generated components. Section 5.8 illustrates the applicability of the framework in a more conventional setting: monitoring for a programming language. Sections 5.9 through 5.11 describe generic monitors: ones that use monitoring interfaces without attaching domain-specific semantics to the elements of those interfaces. A generic monitor views events just as entities with attribute values; in contrast, a domain-specific monitor knows *what* each event means in terms of the problem domain.

Our concentration in this chapter is on the characteristics of the program components that influence the design of monitors and monitoring interfaces. We begin each section with a discussion of a particular program component that we would like to monitor. We then present a monitor design that supports desirable monitoring capabilities for that component. Finally in each section, we consider monitoring interfaces that can be provided by component implementations to support our monitor design.

A central consideration for all of these monitoring interfaces is the storage of relevant information. Some components will generate useful data for monitoring but throw some or all of it away once generated. Others will preserve it as part of the normal function of the program. In the case where data is thrown away, monitors must themselves store anything they will need. The choice between using an event or a data operation in a monitoring interface is largely governed by this need for data storage. If a monitor must store the data itself, events are used to communicate the data in a timely incremental fashion. On the other hand, if the component stores the data as a matter of course, it is

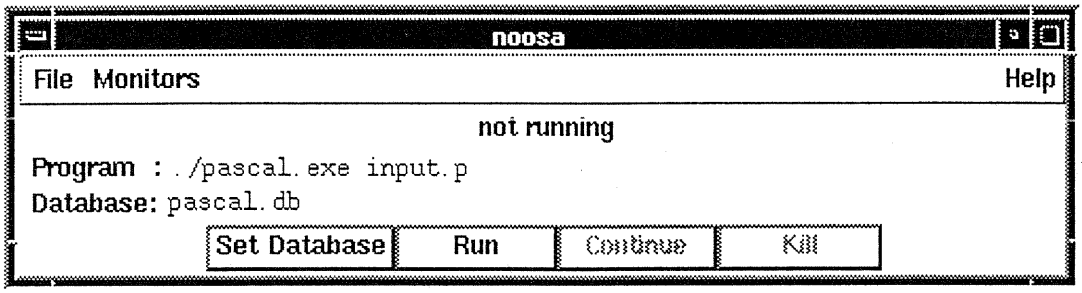


Figure 5.1: A Program Control Monitor.

usually more desirable to access it when needed via data operations to avoid unnecessary duplication. This chapter contains examples of both of these approaches as well as combinations.

Our discussion in this chapter is at the level of the monitoring framework rather than that of any particular framework implementation. The monitoring interfaces described are applicable to any system implementing the framework of Chapter 4. The monitors shown in this chapter were built using the implementation described in Chapter 6.

The Eli examples in this chapter are presented using a typical Eli-generated program as the subject. The program is a compiler for a subset of Pascal called Pascal- (read "Pascal minus") as described by Brinch-Hansen [1985]. The complete specifications used by Eli to generate the compiler are given by Waite [1993]. It is important to note that all of the monitors described here are independent of the specifications used. They are applicable to any Eli-generated program.

5.1 Monitoring Environment

Before we consider specific monitors, it is germane to describe the setting in which they operate. Any monitoring environment must include a program control monitor that is used to specify the program that should be monitored and its command-line arguments. The control monitor we use is shown in Figure 5.1. Subjects can be run, be told to continue execution after they have stopped, or be killed. Their monitoring database can also be specified (see Section 4.4). This control monitor can be built in a straightforward manner using operating system primitives and the predefined `continue` and `kill` operations described in Section 4.3.

There is exactly one control monitor active at any time. All other types of monitor can have zero or more instances active at any one time. Instances are created using the "Monitors" menu in the control monitor (see Figure 5.1). Exactly which monitors can be created depends on the subject being monitored. As described in Section 4.4, monitors use the monitoring database to determine whether they are applicable to a particular subject.

Concurrent instances of different monitors allow the user to examine different components at the same time or relate information from one monitor to a display in another. Multiple instances of a single monitor type enable different aspects of the same component to be examined at once. In conjunction with the ability to detach monitors (see Section 4.2.1), it is also possible to compare monitor output from executions of the subject with different input data.

The overall design philosophy applied in this chapter is that there should be a self-contained monitor for each component that might be a target for monitoring. Thus the user can concentrate their attention on aspects that are relevant to the task at hand, rather than being inundated with information as might be the case with a monolithic monitoring environment. This philosophy also allows environment development to proceed incrementally since each monitor is constructed independently.

5.2 Message Monitors

The first component we consider is a simple one. We show how a monitor can animate the operation of a component that is normally considered to produce static output. A simple event is used to communicate relevant information from the program to the monitor.

5.2.1 The Message Component

Eli-generated programs take text as input, analyze that text, and perhaps produce some text as output. During analysis, messages may be produced for a variety of reasons. Eli provides a fixed message component that implements a messaging service for other parts of the compiler. Messages can be printed immediately or queued for later printing in a listing format with the input text. Each message comes with some message text and is associated with a location in the input text.

Some messages indicate error conditions. Others warn about properties of the text that, while not indicative of errors, might be of interest to the user. To indicate the difference between different kinds of messages, they are assigned *severities* by the components generating them. Messages with severity `INFO` are informational. `WARNING` and `ERROR` severities indicate conditions that the compiler can and cannot repair, respectively. `DEADLY` messages indicate a situation in which processing cannot continue.

5.2.2 Monitoring Messages

A useful way to monitor the Eli message component is to provide access to the stream of messages that are produced by the compiler. A monitor must make apparent the correlation between each message and the input location to which it refers. We have two reasonable alternatives for message display: wait until they have all been produced and present a listing of the input text with the messages, or display the messages as they are generated. The former approach is likely to be sufficient for a programmer *using* an Eli-generated compiler, but Eli users (compiler writers) will often be interested in the

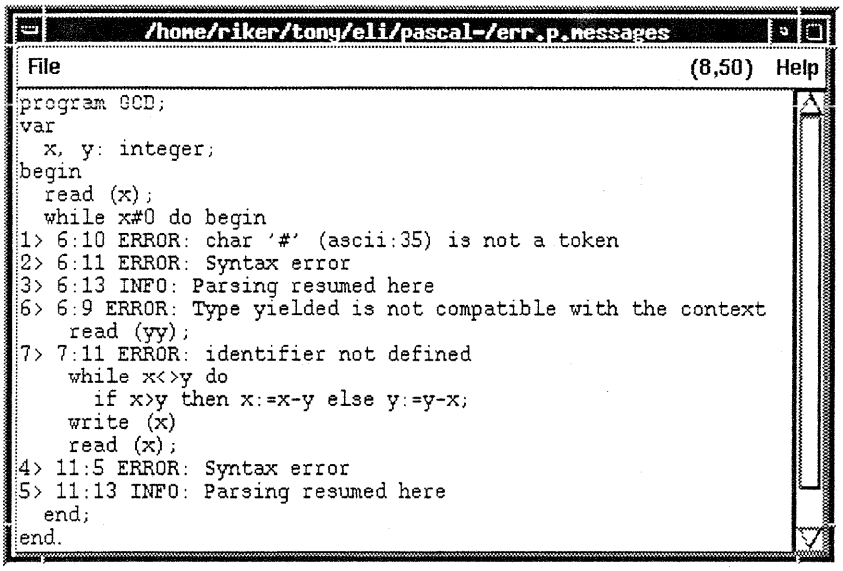


Figure 5.2: A Message Monitor.

time order of the messages. Some messages are produced in response to conditions that have been artificially created by previous parts of the compiler. For example, if a syntax error is found in the type specification of a variable, some compilers will assign a `void` type to that variable. Assignments to that variable may then trigger further messages. Being able to see the order in which messages are produced is particularly important if this order does not correspond to their order in the input text. Thus we design our message monitor so that it displays each message as it is generated.

A message monitor initially displays the input text. As messages are produced by the program they are inserted into the display following the line to which they apply. Thus the monitor implements a dynamic compilation listing. Figure 5.2 shows a message monitor displaying messages from an incorrect version of a Pascal– Greatest-Common-Divisor (GCD) program. Message ordering is given by ordinal numbers displayed at the beginning of each message line.

To allow correlation of the messages with exact source locations, the monitor displays coordinate information near the right-hand end of its menu bar. The coordinate displayed is the line and column of the current pointer (mouse) position in the monitor window.[1]

5.2.3 Monitoring Interface

A message monitor places two demands on the monitoring interface of a program: it needs to get access to the program's input text, and it needs to find out when messages are generated.

The first of these demands must be satisfied by a component other than the message

[1]Some of the other Eli monitors described in later sections display similar coordinate information.

component because the latter does not deal with the input text directly. Eli-generated programs utilize a fixed source component that provides uniform access to input text. Input processing is initialized using an operation that takes the name of the file containing the input text. To enable a message monitor to determine which file is being used, this operation generates an event:

```
event sourceinit "Source file initialization"
   (str filename "File name");
```

A message monitor handles this event by displaying the contents of the named file in its window. Since Eli-generated programs are fundamentally concerned with an input text, a number of the Eli monitors described later also use the `sourceinit` event for this purpose. It is an example of a one-to-many mapping from program component to monitor type.

To satisfy the second demand of a message monitor—finding out when messages are produced—we need another event type. This event will only be used by message monitors. The Eli message component has one routine used by client programs to produce messages. Every time a message is produced, this routine generates a `message` event:

```
event message "Generation of a program message"
   (str severity "Severity of this message",
    str text "Message text",
    int line "Line number",
    int col  "Column number");
```

The handler for message events simply uses the event attribute values to display the severity and message text at the appropriate line and column.

5.3 String Table Monitors

In this section we increase the complexity somewhat over that of message monitors. The component considered is still simple but we design a slightly more sophisticated monitor. Instead of just displaying data from the component we allow the user to interact with that data. To avoid duplicating component data in the monitor we need data operations so that we can get data from the subject when the user wants to see it.

5.3.1 The String Table Component

Most Eli-generated programs need to manipulate text strings. To avoid the overhead of copying the strings around during execution, a string table can be used. Eli has a fixed component that implements a string table allowing integers to be used to represent strings. One operation inserts strings and returns the corresponding integer index. Another looks up strings given their index.

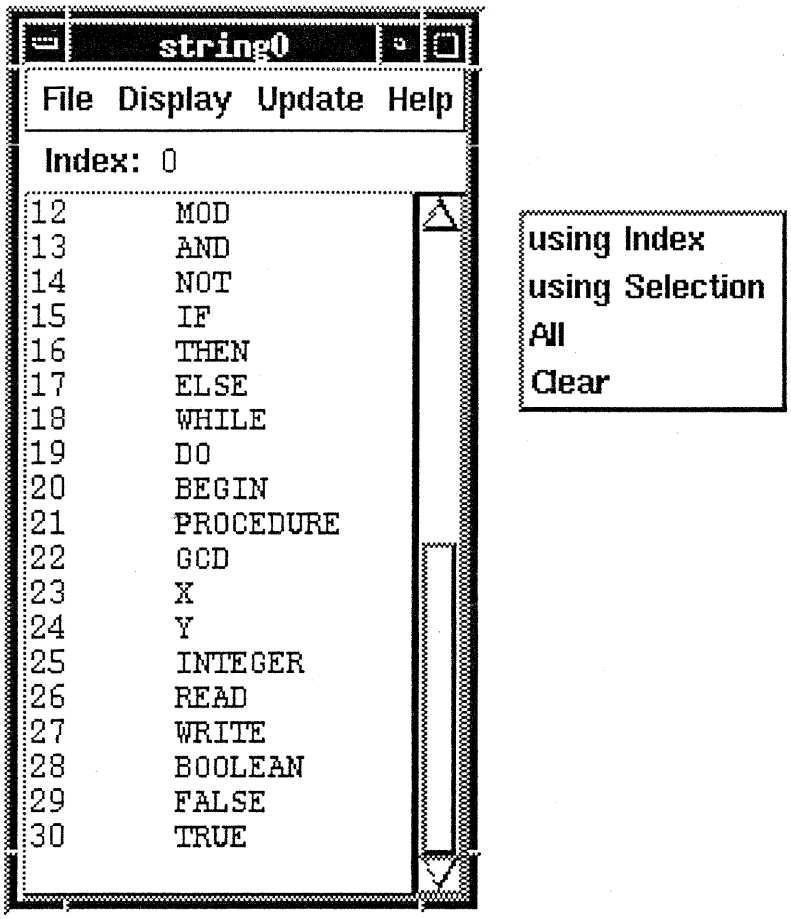


Figure 5.3: A String Table Monitor.

5.3.2 Monitoring the String Table

There are at least three useful ways of monitoring the string table. First, we would like to be able to see which strings are inserted and when they are inserted. We must be able to see the indexes that are assigned to the different strings in order to be able to debug the use of these indexes by other components. Second, it would be useful to be able to get an accurate picture of the current state of the table or some part of it at any time. Our monitor design must make these two modes of operation (and their combination) possible. Lastly, we might like to be able to update string table entries on-the-fly, allowing us to correct for program bugs without recompiling the program.

We can support the first requirement by displaying strings and their indexes whenever they are inserted into the table. Figure 5.3 shows a monitor displaying the contents of the string table once the Pascal- GCD program has been compiled. In this case the string table contains the identifiers and reserved words of the input program. (Only some are visible; the others are accessible using the scrollbar.) Notice that the strings have been uniformly converted to upper case (compare with Figure 5.2). This conversion is

performed by the compiler to enable comparisons that conform to Pascal–'s case insensitivity rules. A string table monitor allows us to verify that the case conversion is being performed correctly.

The second monitoring requirement is satisfied by the monitor as it stands. At any point in the execution, the list of strings currently stored in the table is displayed in the monitor. However, to give some flexibility in display, it is useful to allow editing of the text window displaying the strings. Thus, in a crude way, the user can customize the display to show only strings of interest. By allowing editing, we can no longer guarantee that the displayed strings reflect the complete contents of the strings table. Thus, the "Display" menu (shown at the right of Figure 5.3) contains an "All" item that can be used to display the complete contents.

The "Display" menu provides some other operations: "Clear" allows the display to be emptied quickly and is most useful if the user is only interested in strings inserted after a particular point in time. "using Index" allows particular entries to be examined after their index has been typed into the "Index" field immediately under the menu bar. Similarly, "using Selection" allows a number to be selected with the mouse (in another monitor, say) and the string with that index to be displayed.

Updating string table entries is easily performed by having the "using Index" item in the "Update" menu. It prompts for an index and a string value. The new string is displayed as if the program had inserted it.

5.3.3 Monitoring Interface

Our string table monitor design can be implemented using a monitoring interface that provides: 1) a way to find out when strings are inserted, 2) a way to look up strings at arbitrary times, and 3) a way to update string values.

The Eli string table component has a single operation that is used by components to insert strings. It generates the following event:

```
event string_stored
      "Storage of a new string in the string table"
   (int index "Index of new string",
    str string "New string");
```

A string table monitor handles events of this type by simply displaying the event attributes in its window.

Monitor operations such as "using Index" require the ability to interrogate the table at any time. The string table component provides the following monitoring operation for this purpose:

```
operation get_string "Look up a string given its index"
   (int index "Index of string to be looked up") : str;
```

To implement the "All" menu item it is necessary to have another operation that returns the entire contents of the table.

```
operation get_all_strings
   "Look up all of the index-string pairs" () : str;
```

`Get_all_strings` returns a list of index-string pairs.[2] Note that the implementation is free to use any technique to map indexes to strings. In particular, indexes need not be consecutive or even unique. Just as is the case for functional interfaces, it is important when designing monitoring interfaces to make sure not to reveal too much about component implementations.

The same string table monitor design could be implemented with just `string_stored` events. The monitor could keep its own copy of the string table data by remembering the `string_stored` events. Operations like "All" would then just access the local copy of the data. Instead, we have chosen to use operations to get at strings after they have been inserted, thereby reducing both the effort involved to implement the monitor and the storage the monitor consumes. Tradeoffs between storage consumption and execution time are common in the design of monitors and monitoring interfaces.

Updating string values is supported by the `set_string` operation:

```
operation set_string "Change the value of a stored string"
   (int index "Index of string to be changed",
    str value "New value for string")
```

In addition to actually changing the value, this operation generates a synthetic event (of type `string_stored`) to notify monitors that the value has changed.

5.4 Lexical Analysis Monitors

Lexical analysis provides us with the first situation where the relevant program component does not maintain the data needed for monitoring. The message component keeps a list of messages so that a listing can be produced when the compilation is over. The string table component encapsulates the string storage. In contrast, the lexical analysis component generates data —for the parsing component, see Section 5.5—and throws it away. Our monitor must therefore maintain its own copy of the data. Because all the needed data can be provided by events, no data operations are needed. In this section we also see how it is useful for a monitor to indicate progress during execution. This requirement provides a motivation for the predefined `stopped` event of the framework.

5.4.1 The Lexical Analysis Component

Lexical analysis is the process of converting a stream of input characters into tokens, discarding irrelevant characters such as white-space. Tokens represent literals such as delimiters like ";", or non-literals such as identifiers or numbers. When developing a

[2]The signature of `get_all_strings` says that it returns a string. The framework implementation described in Chapter 6 represents all complex values, such as lists, as strings. Future versions will allow more strongly-typed signatures.

lexical analyzer using Eli, literal tokens are given in a grammar for the input text. For example, a context-free production:

```
WhileStatement : 'while' Expression 'do' Statement .
```

specifies the literal tokens "while" and "do" as part of the Pascal– while statement. In contrast, the appearance of each non-literal token is not fixed but is given by a regular expression pattern. For example, the following specification describes Pascal– identifiers or names as being constructed from a leading letter followed by zero or more letters or digits:

```
Name : $[a-zA-Z][a-zA-Z0-9]*
```

Non-literal tokens usually have an *intrinsic attribute value* associated with them. Intrinsic attributes contain specialized information about the token and enable different instances of a single non-literal token type to be distinguished. For example, integers may have their numeric value as their intrinsic attribute. In the Pascal– compiler, names have their index into the string table as their attribute.

5.4.2 Monitoring Lexical Analysis

Monitoring the lexical analysis process necessarily involves the mapping between characters in the input and tokens. In many cases, errors in later compilation phases such as parsing are due to characters being recognized as the wrong tokens; for example, the floating-point number "5.4" may be incorrectly treated as the integer "5", followed by a "." token, followed by the integer "4", a sequence that may be syntactically illegal. For each token it must be possible to examine the characters from which it is formed, the coordinate (line and column) in the input text where it was found as well as the kind of token that it was determined to be.

To enable the character-token mapping to be accessed, the monitor should first display the input text. Whenever the program is stopped, it should allow the user to select locations in the text and have the monitor display the token (if any) that was found by the lexical analyzer at the selected location. The text of the token should be highlighted, and its type, coordinate information and intrinsic value (if any) should be displayed. The last of these items is needed so that the computation of the intrinsic value can be debugged using another monitor. For example, the string table monitor would be used to check the intrinsic attributes of `Name` tokens in the Pascal– compiler.

A useful property of a monitor is that it should reflect the progress of the subject where appropriate, to provide contextual information about how the program is executing. Monitoring lexical analysis provides an opportunity to illustrate this idea. When the program stops during execution, perhaps at a breakpoint set using another monitor, there will be some prefix of the input text that has been processed by the lexical analyzer; the rest has yet to be processed. The monitor should display the boundary between these two pieces of text so that the user can easily determine how much processing has been done. For example, this progress indication might be useful when using the string table

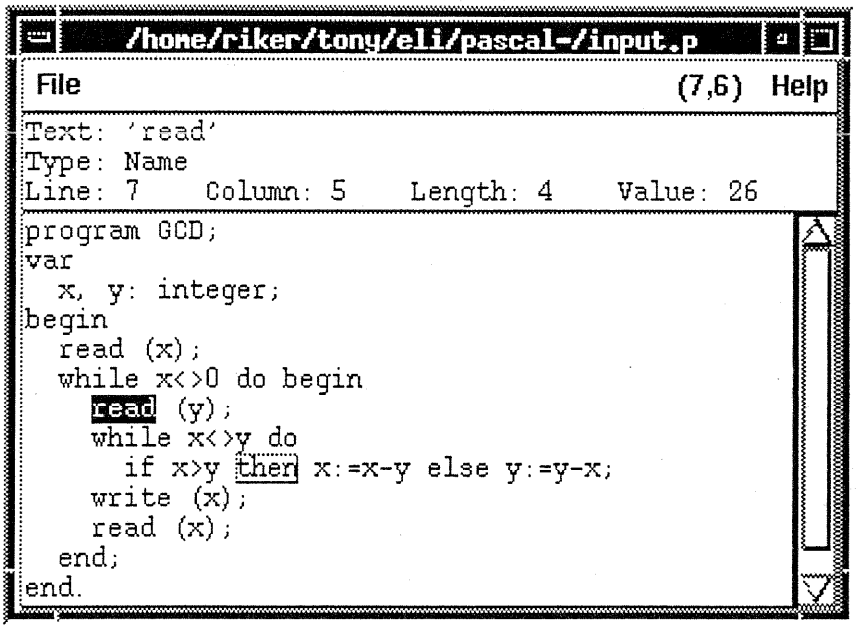


Figure 5.4: A Lexical Analysis Monitor.

monitor. Usually strings are entered into the table during lexical analysis. Knowing the progress of the analyzer enables the user to correctly interpret the output of the string table monitor.

Figure 5.4 shows a lexical analysis monitor monitoring the Pascal- GCD program. The user has selected line seven, column six which is the second character of a `read` token. As a result the monitor has highlighted the token and the top window gives its details.

The token's type is given by name. Non-literal tokens use their name from the lexical analysis specification; literals use their constituent text. The `read` token is an instance of the `Name` non-literal token type. It appears at line seven, column five of the input text, and is four characters long. Its intrinsic value is 26 which can be checked using a string table monitor (see Figure 5.3).

To indicate progress through the input this monitor draws a box around the last token seen. In Figure 5.4 it's the `then` token on line nine. Because the input has not been processed past that point, no tokens after the `then` can be examined.

5.4.3 Monitoring Interface

A straight-forward monitoring interface that supports our lexical analysis monitor contains a single event representing the recognition of a token:

```
event token "Token recognition"
   (str type   "Token type",
    int line   "Line number",
```

```
int col    "Column number",
str lexeme "Character string",
int len    "Length",
int val    "Value",
int code   "Token code");
```

Eli-generated lexical analyzers contain a single routine used to get the next token from the input. This routine can generate `token` events. The monitor stores the event attributes in a list as it is notified of each event. User selection of a text location results in a search through this list testing the extent of each token against the selected location. If an overlap is found, the extent is highlighted and the stored attributes are displayed.

The lexical analysis monitor stores token events for two reasons: 1) it is impractical to display all token information at once in a useful fashion, and 2) the lexical analysis component does not store the token history information itself. It is tempting to get around 2) by modifying the component to keep a record of the tokens that it is generating, but this amounts to putting monitor functionality in the component, which is undesirable.

Our monitor does place a small extra burden on the lexical analysis component. The component must be able to provide a value for the `type` event attribute. Normally a lexical analyzer has no need for the types of tokens; it operates solely with numeric token codes.[3] Thus we see that the monitoring interface requires some capabilities over and above those needed for the functional interface of the component. In this case, the best solution is for Eli to generate a table mapping token codes into token types. At the event generation site for `token` events, the name of the current token is determined from its code using the table.

The lexical analysis monitor uses the predefined `stopped` event (see Section 4.2.2) to provide feedback about progress through the input. Handling a `stopped` event simply involves drawing a box around the last token in the monitor's token list.

An important concern in the design of any monitor is its reliance on the subject's existence. It is useful to be able to perform the following steps during debugging: run the program on some input, detach one or more monitors, create some more monitors, run the program again on some other input, and use the monitors to probe the differences in the program's behavior on the different inputs. Simple monitors like the string table monitor display their complete data in their window. Thus, once they are detached they become complete records of the program behavior (assuming the user has not edited the display) and the monitor need not store data internally. More complex components like lexical analysis require monitors that cannot display all their data at once in a clear fashion. These monitors must therefore store the relevant information for later use. The lexical analysis monitor stores its token list for this reason. Token lists can be examined even if the particular subject execution that produced them has terminated. A similar situation applies to the parsing monitor described in the next section. Of course, the amount of data that must be stored is relevant; some situations demand so much data

[3] `Token` events have a `code` attribute, containing the numeric token code, that is not used by the lexical analysis monitor. It is included in case other monitors that use token codes may want to use it.

that it is impractical to store it all just in case it may be needed. The tradeoffs between space and utility must be examined on a case-by-case basis.

5.5 Parsing Monitors

The parsing component described in this section presents a similar situation to that of the lexical analysis component: it does not store most of the data needed by the monitor. Thus the monitor must store some of the data itself. Other static data is accessible via data operations so this monitor illustrates a hybrid approach to data management.

5.5.1 The Parsing Component

Parsing is the process of determining the structure of an input text given a stream of tokens produced from that input text by a lexical analyzer. In Eli the allowable structures can be given by a context-free grammar, called a *concrete grammar* because it expresses the concrete realization of language constructs in the input text. For example, the context-free rule:

```
WhileStatement : 'while' Expression 'do' Statement .
```

expresses the structure of a Pascal– while loop. `Expression` and `Statement` identify parts of the while loop; their structure is described by other rules.

Eli uses a parser generator to transform the user's concrete grammar into the parsing component or *parser* of the generated program. The parser obtains tokens using the lexical analysis component and matches them against the grammar. The matching process results in a *parse tree* representing the input. Each node in the parse tree represents an instance of a concrete grammar production, a place where that production matched the input text. Each tree node has an associated *textual extent*, which is the piece of the input represented by the node.

For example, the Pascal– fragment:

```
while x<>y do x:=x+1
```

matches the production for `WhileStatement` given above. The parse tree node representing this match will have two children, representing the matches of the `Expression` "`x<>y`" and the `Statement` "`x:=x+1`". The extents of these tree nodes are straightforward to determine. The `WhileStatement` node extends over the entire fragment, while the `Expression` and `Statement` children extend over characters 7 through 10 and 15 through 20, respectively.

5.5.2 Monitoring Parsing

Debugging a concrete grammar is primarily a matter of examining the correspondences between the input text and productions. Most problems with a grammar show up as

syntax errors in legal input or the absence of errors in illegal text. Assuming the user knows a priori whether a given input text is legal or not, a parsing monitor must let him or her determine why the parser is signalling a spurious error or failing to diagnose one. The monitor must therefore be able to display the parse tree corresponding to the input. The mapping between input text and grammar productions must be available for examination.

Note that Eli-generated programs do not actually construct a physical representation of the parse tree; it is just represented by the sequence of grammar productions recognized by the parser. Nevertheless, when designing a concrete grammar, an Eli user will conceptualize the structure of the program as a tree. Thus the parsing monitor should support this view.

Our parsing monitor design calls for the input text to be displayed as was done for lexical analysis monitoring. Selecting a location in the input text causes the monitor to display the grammar productions (if any) that were used to recognize that location. Figure 5.5 shows a monitor after the user has selected the `x` name on the left-hand side of the `x:=x-y` assignment.

Productions in the upper window go from most general at the top to most specific at the bottom. Thus the first production in the top window is the root of the grammar. The others represent a path in the parse tree from the root to the most-specific node representing the selected location. In this case the productions identify the `x` as a `Name` which is a `VariableNameUse` inside a `VariableAccess` in an `AssignmentStatement`, and so on. The underlined symbols in all but the last production denote the left-hand-side symbol of the next production. For example, the production for the `IfStatement` shows that the following `Statement` is in the then-clause rather than the else-clause.

Selecting a production instance in the upper window highlights the extent in the input text that was recognized by that production instance. Figure 5.6 shows the result of selecting the `IfStatement` instance.

5.5.3 Monitoring Interface

Parsing monitors with this design can be used with either of the two parser generators available in Eli. To be able to support monitoring, the generated parsing components must notify the monitor when pieces of input text are matched with productions. The monitoring interface contains the following event:

```
event recognition
      "Recognition of a production during parsing"
   (int prod "Number of production",
    int uses "Number of preceding recognitions subsumed",
    int linebeg "Line number of beginning extent",
    int colbeg "Column number of beginning extent",
    int lineend "Line number of ending extent",
    int colend "Column number of ending extent");
```

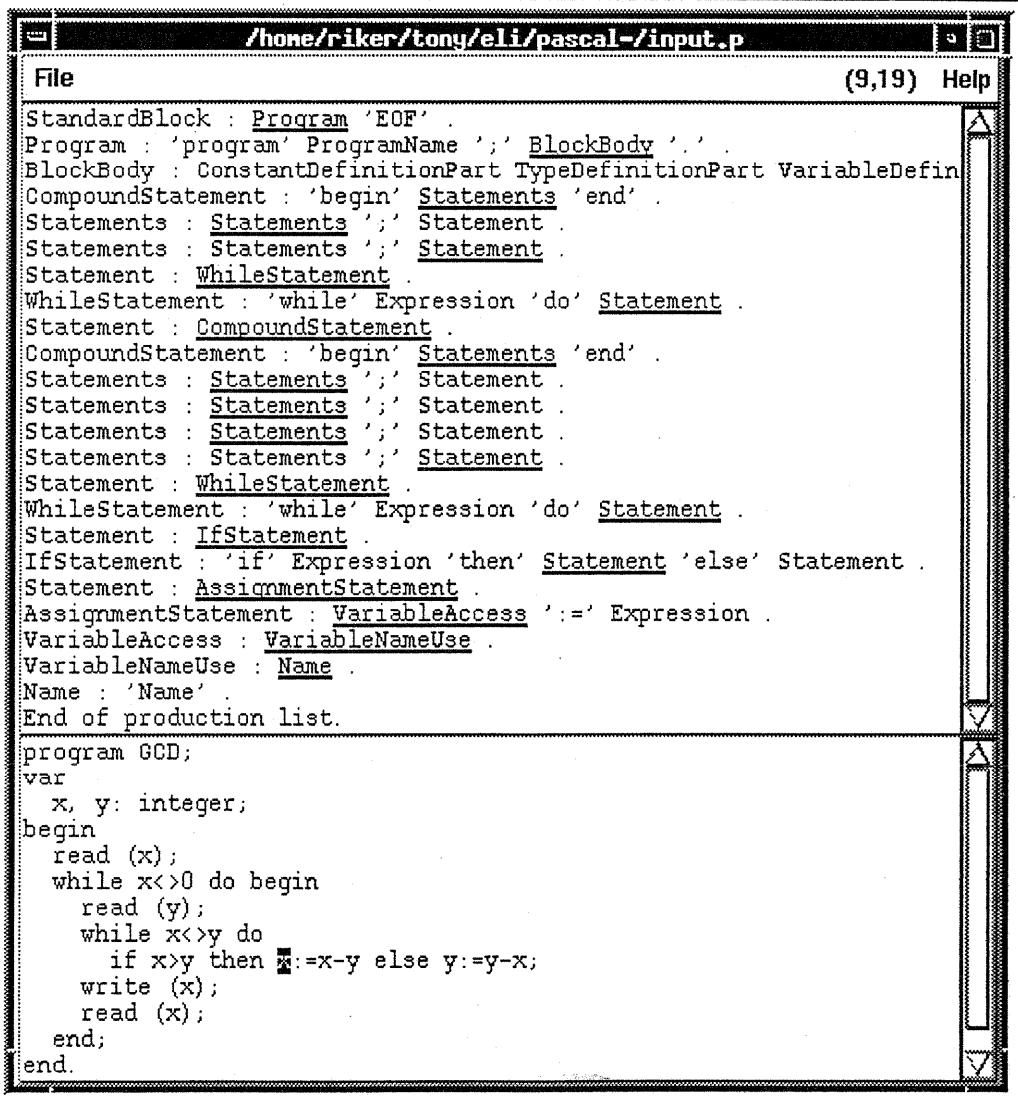


Figure 5.5: A Parsing Monitor: Selecting Input Text.

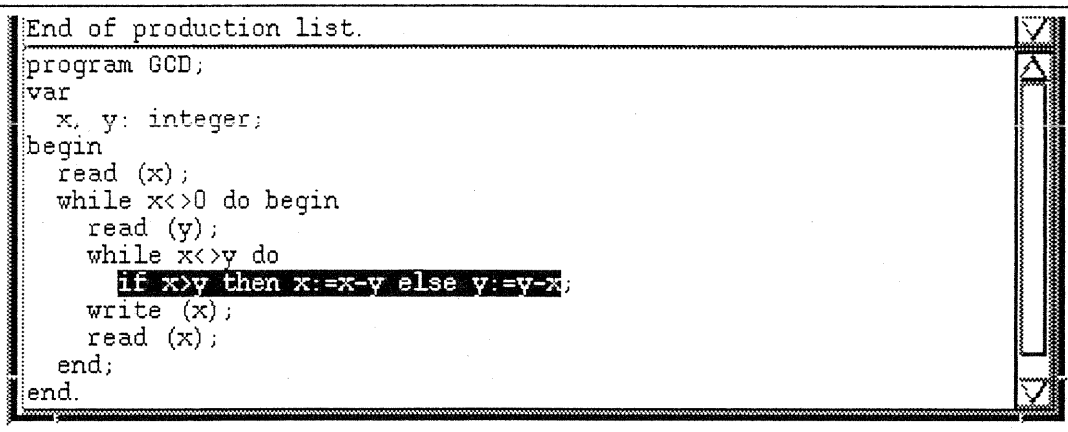


Figure 5.6: A Parsing Monitor: Displaying Production Instance Extent.

`Recognition` events are generated whenever the parser matches a piece of the input text with a production. The `prod` attribute identifies the production. The last four attributes identify the input text extent.

A data operation lets the monitor map a production number into the text of the production as supplied by the user specifications:

```
operation get_conc_prod
      "Look up the text of a concrete production"
   (int index "Index of the production") : str;
```

Generated parsing components supporting monitoring clearly have to encapsulate more data than they would if monitoring were not desired. Normally the parser does not need to keep track of the end of each extent; the beginning is sufficient to enable reasonable error messages to be issued. Similarly, the texts of productions are usually not of use once the parser has been generated.

Our parsing monitoring interface illustrates a design decision concerning the subject's existence similar to that encountered with the lexical analysis monitor. Sequences of `recognition` events are equivalent to the concrete representation of the input text. Because Eli-generated programs do not themselves keep a representation of the parse tree, it is up to the monitor to do so. In the current design, parsing monitors only store the event attributes; the texts of productions are not stored because, in a typical grammar, productions can be quite long and many will not be referenced during a debugging session. In cases where the output of successive runs of a single subject are being compared, the `get_conc_prod` operation can be used by both monitors without any problem.

A different system might have parsing components that do keep the parse tree explicitly. Such a system might use a monitoring interface that supports operations to access that tree instead of transmitting the information via events. This kind of interface trades off ease of monitor implementation with some added complexity in program components and storage consumption. The next section presents a similar interface to the major data structure used by Eli-generated programs: the abstract syntax tree.

5.6 Attribution Monitors

This section describes the monitoring for the central component of most Eli programs. Consequently, the monitoring interface is the most complex considered in this chapter. It is convenient to view the operation of this component as having a control portion and a data portion, so we describe separate monitors for these pieces. Communication between the two kinds of monitors takes place using a *synthetic event* introduced specially for this purpose and never generated directly by the program.

The main data structure used is maintained by the subject, so the monitors mainly use a *browsing interface* to avoid duplication. That is, they use data operations to browse the data structure without necessarily ever reading the whole thing. Some events are used to enable execution control based on the operation of the component. Use of an event translation is also illustrated.

5.6.1 The Attribution Component

Eli-generated programs typically implement much of their semantic analysis using attribute grammars. An Eli user supplies attribution rules operating on an *abstract syntax tree* structure. In contrast to a parse tree, an abstract tree usually reflects a structure appropriate for attribution rather than (necessarily) one appropriate for parsing.[4] Eli automatically arranges for the abstract syntax tree corresponding to the input text to be built during parsing.

Attribution is given in terms of productions. Each computation attached to a production is applied at all places in the abstract tree where that production was applied. For example, the following attribution is performed for each node that represents an if-statement with no else-clause:

```
RULE OneSided:
   IfStatement ::= 'if' Expression 'then' Statement
COMPUTE
   Expression.ExpectedType = BooleanKey;
END;
```

It specifies that the expected type of the expression part of an if-statement is boolean. Other attribute computations use the `ExpectedType` attribute. For example, attribution for rules with `Expression` on the left-hand-side compares the expected type with the actual type and issues an error if the two are not compatible according to the type rules of Pascal–.

The dependencies between attribute computations define a partial ordering among those computations. Eli transforms an attribute grammar specification into an *attribute evaluator* that is guaranteed to evaluate the attributes in a correct order for any tree conforming to the given abstract syntax. The evaluator also automatically allocates

[4]The chief differences between the concrete and abstract syntaxes for conventional programming languages arise from operator precedence and associativity in expressions.

storage to hold the computed attribute values throughout their lifetimes. Often attribute lifetimes do not overlap, so storage can be reused—a significant saving for large grammars.

Frequently it is necessary to introduce attributes whose sole purpose is to control the ordering of other attribute evaluations. They are called *void attributes* because they have no value. For example, a grammar may have a void attribute indicating that all of the declarations within a block have been processed. Processing of uses of the declared entities would depend on this void attribute to ensure that all relevant information was available. Thus in this case the void attribute serves to order processing of declarations and uses. Attribute evaluators built by Eli conform to dependencies introduced by void attributes. Since void attributes have no values, no storage is allocated for them.

5.6.2 Monitoring Attribution

Since attribution is based on the abstract syntax tree, an attribution monitor must display the tree in some fashion. It is sometimes the case that problems in attribution result from having the wrong tree structure. As was the case for the parse tree, each node of the abstract tree has an extent in the input program that it represents. Our monitor should make this extent information available.

The interaction between different attribute computations can be complex. Because computations are applied in contexts determined by the particular abstract syntax tree derived from the input text, it is easy for attributions from different rules to interact in unexpected ways. A second requirement of any attribution monitoring tool must therefore be to allow the actual values of attributes to be examined. The abstract tree display should enable the user to designate which attribute values are of interest.

Attributes express important conditions that hold during attribution. Conditions expressed by attributes are useful during monitoring because they break up semantic analysis into meaningful phases. For example, it may be important to examine the program state after all declarations have been processed but before uses have been examined. If a void attribute has been used to express this condition, we should be able to use it to control execution. Thus another requirement of an attribution monitor is that it allow execution to be stopped when the conditions expressed by attributes hold (whether or not the attribute holds a value).

Conventional attribute grammars only allow attribute computations to access attributes of the symbols appearing in the rule where the computation is placed. In many cases attributes of nodes that are further afield are useful. Eli permits attribution to refer to remote attributes using shorthand notations: The `INCLUDING` construct allows reference to an attribute of a node higher up (closer to the root) in the tree. For example, the following is a typical construct used to find the environment from which to obtain the meaning of an identifier:

```
INCLUDING (Program.Env, Block.Env)
```

Here we assume that the non-terminals `Program` and `Block` each represent the beginning of new identifier scopes. This `INCLUDING` construct will return the `Env` attribute of the

closest `Program` or `Block` node above the current node in the tree. The value of this attribute represents the scope containing the current node. Given the capabilities of our attribution monitor discussed so far, it is possible for the user to manually examine the tree to find the sources of remote attribute values and examine them there. However, requiring this effort goes counter to our principle of allowing the user to monitor in terms of their specifications because it forces them to duplicate the behavior of the algorithm used to implement a remote access. A better situation would be to allow the user to examine the value of an `INCLUDING` construct at the node where it is used, as if it were a regular attribute value.

It is appropriate at this point to consider how a source-level debugger performs as an attribution monitor. Stopping when attribute conditions hold involves setting a source-level breakpoint at the appropriate point in the attribute evaluator. Determining this point can be difficult and involves a great deal of knowledge of the translation performed by Eli. This translation can be complex, particularly where remote attribute accesses are concerned. Examining an attribute value involves stopping at a point within its lifetime and looking at its storage. Determining the lifetime of an attribute and where it is stored is a non-trivial task requiring detailed knowledge of the attribute evaluator. If these problems were not enough, consider that small changes to the input attribute grammar can change the resulting attribute evaluator considerably. Changes to attribute computations can even invalidate previously obtained information for attributes whose computations have *not* changed. Our conclusion is that using a source-level debugger to monitor an attribute computation is prohibitively difficult.

The attribution monitor design described in the rest of this section allows the abstract syntax tree to be examined and attributes of tree nodes to be designated of interest. When the program is run it will stop execution whenever an interesting attribute value is computed. For void attributes, the program will stop at the point where the condition expressed by the attribute is first established.

Attribute values can be examined using a separate *value monitor* that will display each interesting attribute value as it is computed. Keeping the value monitor distinct cleanly separates the two monitoring operations of execution control and value examination.

Consider the following trivial Pascal– program:

```
program range;
type myarray = array [ 0 .. 42 ] of integer;
begin end.
```

Figure 5.7 shows an attribution monitor displaying the part of the abstract syntax tree corresponding to the `myarray` type definition. Each node is displayed with a unique node number, the name of the non-terminal represented by the node, and (in parentheses) the name of the abstract rule applied at that node to derive its children. The tree structure is indicated by indentation. Selecting node twelve causes it to be highlighted and the production `ArrayDef` (which is applied at that node) to be displayed in the window immediately above the tree. The extent of the node is also highlighted in the input text window below. To enable rapid navigation in the tree, selecting an input text location highlights the node that derives that location.

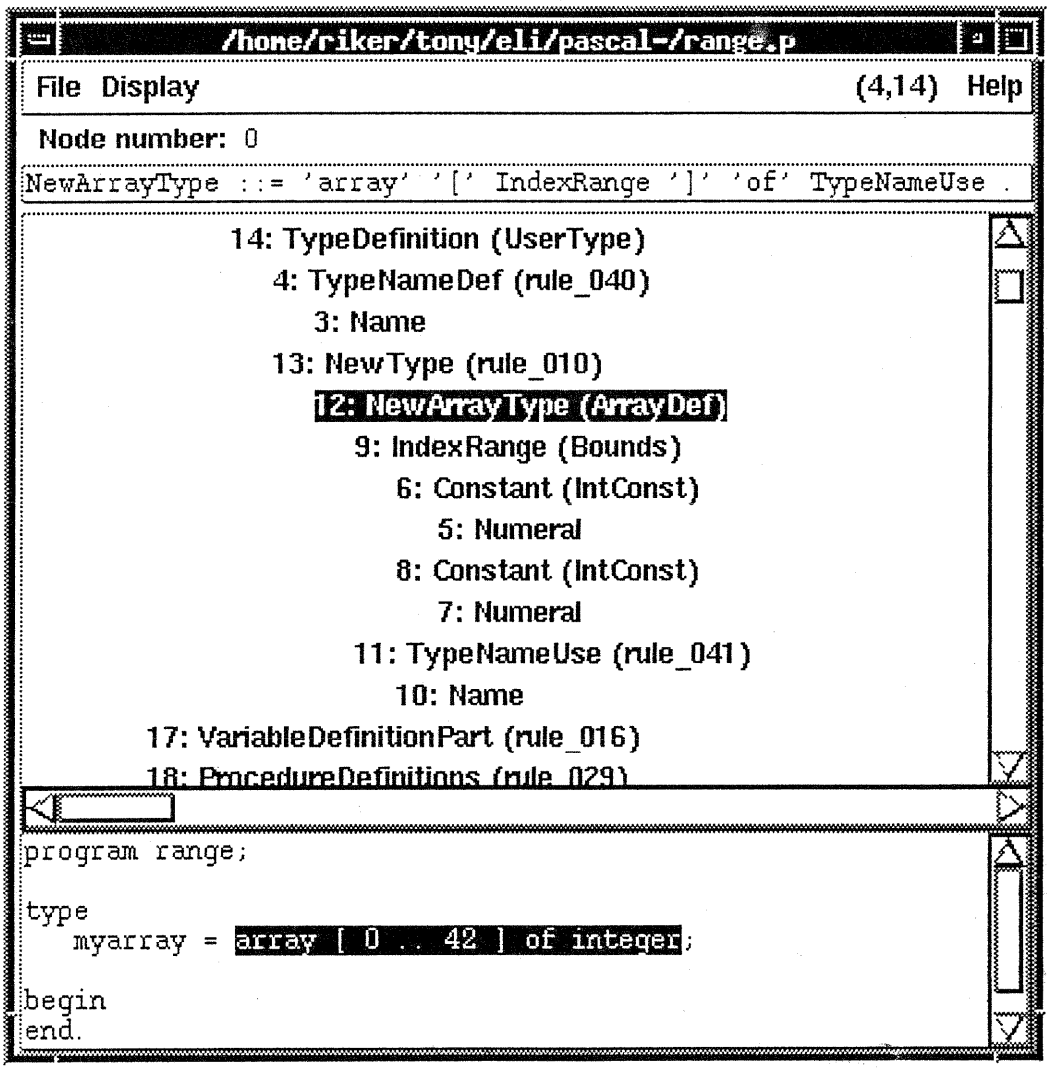


Figure 5.7: An Attribution Monitor.

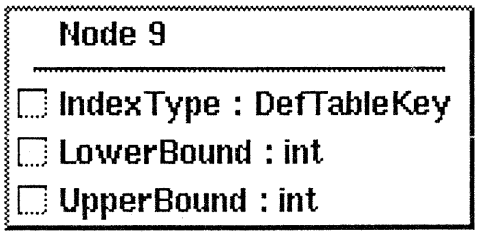


Figure 5.8: An Attribute Menu.

At each node a set of attributes can be selected via a pop-up menu. For example, the `IndexRange` non-terminal has two attributes `LowerBound` and `UpperBound` representing the particular range denoted by the non-terminal. They are computed from attributes of the constants of the range with the following attribution:

```
RULE Bounds:
   IndexRange ::= Constant '..' Constant
COMPUTE
   IndexRange.LowerBound = Constant[1].Value;
   IndexRange.UpperBound = Constant[2].Value;
END;
```

Figure 5.8 shows the menu for an `IndexRange` node (node nine in Figure 5.7). This node also has another attribute, `IndexType`, a definition table key, which is defined by other attribution on this rule.

In an attribute menu, each attribute is associated with a check button. Checking an attribute indicates interest in that attribute. When the program is next run, execution will stop when that attribute is computed. The relevant node is also highlighted. When the user selects an attribute while the program is stopped during execution, the attribute's value may have already been computed. In this case, the program will need to be run again from the beginning for this attribute selection to affect execution.

If the programmer wishes to examine the value of the attribute, a value monitor must be used. Whenever an interesting attribute value is computed, the name of the attribute, its type and its value will be displayed in the value monitor. Figure 5.9 shows the value of the two bound attributes of node nine, allowing us to confirm that they have been computed correctly.

In contrast to source-level debuggers, the attribution and value monitors allow painless monitoring of attribution. All interaction with the monitor is performed in terms familiar to the user: the structure of the abstract syntax tree and the names of attribute values. No knowledge of the implementation of the attribute evaluator is needed.

5.6.3 Monitoring Interface

The monitoring interface used by attribution monitors and supported by Eli-generated attribute evaluators provides three different event types and a number of operations.

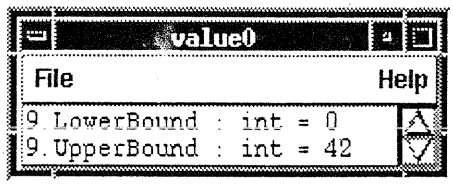


Figure 5.9: A Value Monitor.

A `tree` event is used to indicate that the abstract syntax tree has been constructed by the parser and hence can be displayed.

```
event tree "Abstract tree has been constructed"
   (int root "Handle for the root of the tree");
```

In order to display the tree structure, attribution monitors use a variety of operations. The abstract tree represents a sizable amount of information and it is rare for the programmer to need to examine every piece of that information: all the tree nodes, or all the attributes of each kind of tree node. None of the tree structure is explicitly represented in the monitor. Instead the monitor provides a browsing interface to the tree and details are obtained from the subject when needed. Typical examples of operations used for this purpose are `get_node_prod`, which allows rule applications to be identified, `get_children`, which accesses the tree structure, and `get_node_attrs`, which returns the attributes of a given node:

```
operation get_node_prod
      "Look up the rule represented by a node"
   (int node "Handle for the node") : str;
operation get_children
      "Return the handles for the children of a node"
   (int node "Handle for the parent node") : str;
operation get_node_attrs
      "Return a list of attributes for a node"
   (int node "Handle for the node") : str;
```

The implementation of these operations is a straightforward translation of each request into an operation on the tree structure maintained by the evaluator.

When attributes are computed, `attribute` events are generated:

```
event attribute "An attribute value has been computed"
   (int node "Node to which attribute applies",
    str attr "Name of the attribute",
    int value "Value that was computed");
```

The Eli tool that produces the attribute evaluator inserts event generation code at appropriate places in the evaluator.

Not every `attribute` event corresponds to an interesting attribute. Attribution monitors therefore use tests in handlers for `attribute` events to detect the interesting ones. When an attribute of interest is computed, the handler will generate an `attribute_value` event to signal to value monitors that a value should be displayed:

```
event attribute_value
      "An attribute value is ready for display"
   (int node "Node to which attribute applies",
    str attr "Name of the attribute",
    int value "Value that was computed",
    str type "Type of the attribute value");
```

Value monitors simply display the attributes of any `attribute_value` event they see. `Attribute_value` events are synthetic events because they are not generated directly by the program itself.

Attribution provides an example of the use of event translation. Recall that event translation is needed because we do not want components to reveal their internal implementation details via events. Eli implements remote attribute accesses, such as applications of the `INCLUDING` construct, by generating regular attributes that *transport* the value from its source to the location of the `INCLUDING`. Since they are treated as regular attributes, these transport attributes will cause `attribute` events to be generated. The event stream thus reveals the implementation of the `INCLUDING` construct. To keep it hidden, we specify a translation for `attribute` events that renames the transport attributes:

```
translation rename_includings attribute
   "Translation for INCLUDINGs"
{
   if attr is a generated attribute for an INCLUDING then
      attr = map_to_including (attr);
   end
}
```

In this pseudo-code, `attr` refers to the event attribute of the `attribute` event type. The mapping from generated names to `INCLUDING` expressions (`map_to_including`) is provided by the Eli internal tool that performs the expansion of includings into transport attributes. The net result is that instead of having the generated attribute name, any applicable `attribute` events will have as their name the `INCLUDING x.y` expression actually used in the attribution. The value of the `INCLUDING` can then be displayed in value monitors as is done for regular attributes.

5.7 Attribute Value Browsers

Browsers are monitors that present views of the execution state of the subject (Delisle, Menicosy and Schwartz [1984]). Attribute value browsers present interfaces to the state

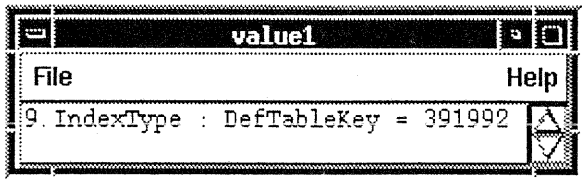


Figure 5.10: A Value Monitor Showing a Complex Value.

of the attribution component of an Eli-generated program. Thus browsers primarily use data operations to access the data structures they are browsing. However, browsers also introduce another important technique: the adaptation of monitors to dynamic data in different executions of the program.

5.7.1 Complex Attribute Values

Many attributes have simple values such as integers or booleans. However, attributes with more complex values are also common. For example, consider again the `IndexRange` non-terminal from our examples above. We saw in Figure 5.8 that this non-terminal has an `IndexType` attribute of type `DefTableKey`. This type is used for values that are keys into the compiler's definition table: a table used to store arbitrary properties of entities. The `IndexType` attribute stores the key of the type to which this particular range belongs. In Pascal–, this type is the same as the type of the constants used to specify the range. Eli programs that use the definition table contain a generated property-key component responsible for maintaining mappings from keys to properties.

Another type of complex attribute value appearing in many programs generated by Eli is that used to store environments. This type allows the representation of nested name spaces. Values of this type are conventionally used to apply the scoping rules of the language to map identifier usages to the objects that they denote. Environments are associated with abstract syntax non-terminals that represent different scoping levels, and they map identifiers to definition table keys. For example, in the Pascal– compiler the scope of a complete program is represented by the `Env` attribute of the `Program` non-terminal. A fixed Eli component provides operations for manipulating and querying environment values.

5.7.2 Monitoring Complex Values

A definition table key is implemented as a pointer to a data structure holding the properties of the entity represented by the key. Knowing the value of the pointer is not sufficient to monitor the property values because monitoring requires knowledge of the details of the property data structure. Thus, value monitors are not sufficient because they only know how to display simple numeric data. Figure 5.10 shows a value monitor displaying the `IndexType` attribute of node nine from Figure 5.7.

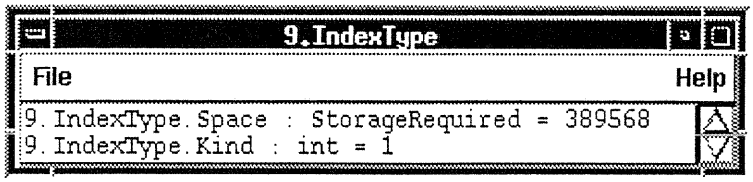


Figure 5.11: A Key Browser.

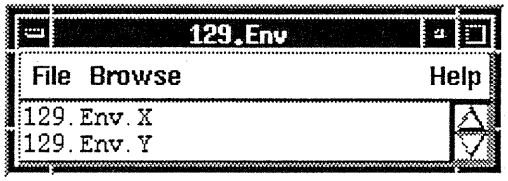


Figure 5.12: An Environment Browser.

Attribute value browsers are one mechanism for allowing the display of complex values. We allow the user to select a complex value in a value monitor to create a browser on that value. Browsers for complex values are implemented in the same way as other monitors via support in the component that implements those values.

A browser for key values would display the properties of the entity represented by the key. Figure 5.11 shows the properties for our example `IndexRange.IndexType`. The properties are displayed with their types and their values. Note that the `Space` property is another complex property, so we could allow the user to select it in the key browser to bring up another kind of browser to examine its value.

Figure 5.12 shows a similar browser for environment values. It is browsing the contents of the program environment of the GCD program (see, for example, Figure 5.6) when execution has completed. Selecting one of the identifiers in an environment browser will create a key browser on that identifier's key. The `Browse` menu allows access to the environment in which this one is nested. In this case, the parent environment is the standard Pascal- environment providing predefined identifiers such as `TRUE` and `FALSE` (Figure 5.13).[5]

Another design point for attribute value browsers is not apparent from a static view but arises when the dynamic use of the monitoring environment is considered. An attribute value browser is created by selecting an attribute value in a value monitor. It uses the actual value of the attribute to perform browsing. Consider what happens when the program is run again from the beginning. Since many complex attribute values are built using dynamic memory allocation it is highly likely that an attribute will not have the same value from run to run. Thus it is a desirable property for a browser on such a value to not only work with the current value but to adapt to new values of the attribute

[5]We use the notation uparrow-x to indicate the parent environment of the environment stored in attribute value x.

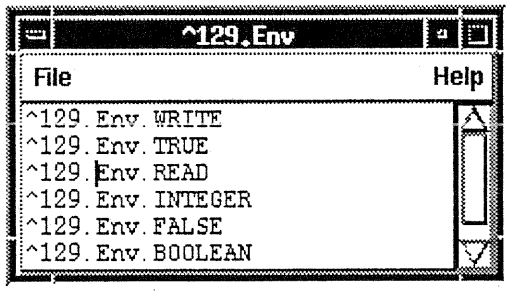


Figure 5.13: The Pascal- Standard Environment.

in successive runs. This relieves the user from the burden of recreating browsers each time the program is run.

5.7.3 Monitoring Interface

Key browsers access the property selector-value pairs for a given key using the following operation:

```
operation get_prop_pairs
      "Look up all selector-value pairs for a key"
   (int key "Key of entity") : str;
```

Selectors are numeric encodings of property kinds. Given a selector, its name and type are obtained with the following:

```
operation get_selector_name
      "Return the name of a numeric property selector"
   (int selector "Selector of interest") : str;
operation get_selector_type
      "Return the type of a numeric property selector"
   (int selector "Selector of interest") : str
```

The Eli environment management component provides two operations that support the browser. The first returns all the identifier-key pairs stored in a given environment. The second gives access to the nesting structure of environments.

```
operation get_env_pairs
      "Look up all ident-key pairs from an environment"
   (int environ "Environment of concern") : str;
operation get_enclosing_env
      "Look up the parent of an environment"
   (int environ "Environment of concern") : str;
```

Browsers on attribute values will update their displays when those values are changed. Note that the actual values of the attributes themselves can never change, but the complex values that they denote can. For example, any identifiers added to an environment during compilation will be added to any browsers displaying that environment. The following event is used:

```
event ident_entered
      "Entering an identifier into an environment"
   (int environ "Environment into which ident was entered",
    int ident "Identifier that was added",
    int key "Key associated with the identifier");
```

Similarly, properties added or changed will be reflected in key browsers upon generation of the following event:

```
event property_set "A property of an entity has been set"
   (int key "Key of entity",
    str prop "Property being set",
    str type "Type of property",
    int value "New value for property");
```

The operations and event generation needed for the attribute value browsers are supported in a straight-forward fashion by accessing the data structures maintained by the property-key and environment components.

Finally, we consider the dynamic adaptation of the browsers to new attribute values in successive runs of the program. A browser "knows" which attribute it is browsing so it can watch for `attribute` events that describe that attribute. When it sees an applicable event, it changes the value it is browsing to match the new value, and updates its display. This mechanism is a simple distributed solution to the problem of keeping monitors up-to-date throughout a monitoring session that may include many runs of the program.

5.8 Monitors for Very High-Level Languages

Traditionally, execution monitors have been used to monitor programs written using conventional programming languages. In this sense, the Eli monitors described above are atypical because they deal with programs written in high-level specification languages such as context-free grammars. This section shows that our execution monitoring framework applies equally well to a more conventional setting.

Many programming languages provide very high-level facilities to programmers. Concise notations allow powerful operations to be expressed succinctly. We can regard these languages as allowing the reuse of the components that implement these notations. A compiler for a language takes a specification of how the components are to be used (the source program) and produces an executable program including the implementation of

those components that are needed (run-time support). This section presents an application of domain-level monitoring in the setting of a very high-level programming language. The monitor illustrates how our framework can be used to provide a simple animation of program execution that allows backward execution and replay.

5.8.1 Icon

Icon (Griswold and Griswold [1983]) provides a host of high-level facilities. Strings, sets, lists, and tables are all first-class objects supported by a large number of built-in operations and operators. The run-time support for these types and operations is considerable, much more than that needed for lower-level languages such as C (Kernighan and Ritchie [1978]) that primarily utilize types directly supported by machine hardware.

As an example of Icon's high-level facilities, consider its string scanning operations. The construct:

string ? *expression*

applies the scanning operations in *expression* to *string*. Within *expression* the current scanning position is maintained implicitly; it is initialized to the leftmost end of *string*. The `tab` operation allows the scanning position to be moved and returns the characters of the string between the old and new positions. Operation `upto` returns the position before the next occurrence of any character in its argument set. Similarly, `many` returns the position before the next occurrence of a character *not* in its argument set. For example, the following code fragment prints words consisting of characters from the set `wordchars` from the value of the variable `line`:

```
line ? while tab (upto (wordchars)) do
   write (tab (many (wordchars)))
```

5.8.2 Monitoring String Scanning

Our framework can be used to provide monitoring facilities for very high-level languages in general and Icon in particular. Debugging string scanning operations can be difficult because the current scanning position is implicit. Changing code to print out the different positions can perturb the code significantly and is error-prone. In contrast a component-level monitor can easily provide intuitive visual debugging.

Figure 5.14 shows a simple monitor that displays the string being scanned and the current scanning position. The right of the menu bar displays the most recently executed operation that moved the position plus the resulting position.[6] As the position is changed the arrow moves. The `Backward` and `Forward` buttons can be used to retrace previous position changes. That is, they provide a general replay facility within the current scanning operation.

[6]Icon string positions can be expressed in both positive or negative forms. The positive form counts up from the left of the string starting at 1. The negative form counts down from the right of the string starting at 0.

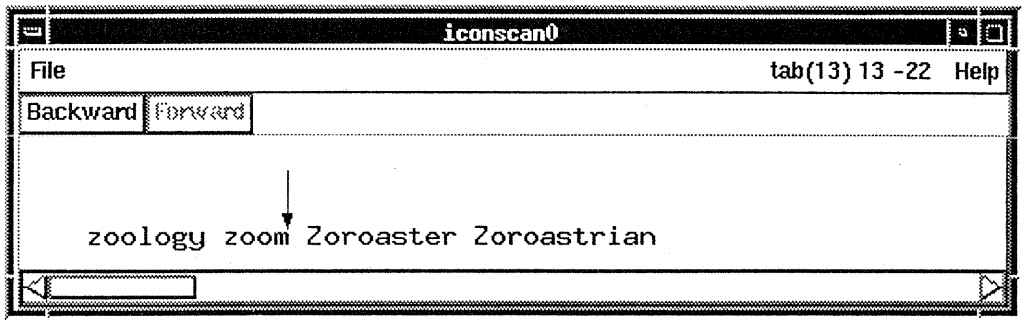


Figure 5.14: An Icon String Scanning Monitor.

5.8.3 Monitoring Interface

String scanning monitors rely on the `scan` and `setpos` events. `Scan` events are generated when execution of a string scanning expression begins:

```
event scan "Begin scanning a new string"
   (str string "String being scanned",
    int length "Length of the string");
```

They are handled by displaying the string and resetting the current position arrow to the left of the string as dictated by the semantics of the string scanning operation.

`Setpos` events indicate the progress of the current position during scanning as well as giving reasons for any changes:

```
event setpos "A movement during string scanning"
   (str oper "Operation performing the movement",
    int arg "Argument to operation",
    int pos "Position being moved to");
```

Each `setpos` event moves the arrow, displays the details of the movement in the title bar, and records it in a history list for possible use by the `Backward` and `Forward` buttons. The monitor maintains history back to the last `scan` event.

The run-time system for the Icon Compiler (Walker [1991]; Walker and Griswold [1992]) was modified to generate these events at appropriate times. Support for a full-featured Icon monitoring environment could be built along similar lines. Thus this exercise provides evidence that the framework is easy to integrate with programming language implementations.

5.9 Generic Monitors

The monitors described in this chapter up to this point are domain-specific; they deal with program components from particular problem domains (compilers and Icon string

scanning) and apply domain-specific interpretations to the information obtained through monitoring interfaces. For example, a few of the Eli monitors use line and column numbers to relate internal data such as tokens to the input text of the compiler. Components produce values for line and column number event attributes which are interpreted by the monitors to enable (say) highlighting of text.

In contrast to domain-specific monitors, *generic monitors* apply no interpretation to the data that they get from the subject. Generic monitors tend to provide facilities that are similar in nature to those provided by source-level tools because the latter are similarly constrained to a generic view of program data. When using a source-level debugger, it is possible to compare values of variables at the level of the programming language but one cannot obtain domain-based views of data without extra work. Generic monitors enable examination of raw monitoring data to complement the higher-level interpretation of that data provided by domain-specific monitors.

Even though they deal with data at a low level, generic monitors can be powerful. The next section describes the design of a breakpoint monitor that controls the execution of the subject. Section 5.11 describes how profilers can be constructed using our framework. Our examples use the Eli domain, but these monitors can be applied to any problem domain that supports the monitoring framework.

5.10 Execution Control: Breakpoints

In source-level debugging tools, *breakpoints* (Johnson [1982]) are usually used to control the execution of the program being debugged. *Code breakpoints* are specified in terms of source code entities like function names or line numbers. When the program calls the specified function or reaches the given line number, the breakpoint triggers and execution stops. Some debuggers support different kinds of *data breakpoints* which 1) trigger when a piece of data is modified in any way, 2) trigger when the value of an expression changes (sometimes called *watchpoints*), or 3) trigger when the value of an expression becomes true (assertion-driven breakpoints). Data breakpoints contrast with code breakpoints in that they assert global properties of program data independent of the current locus of control in the program. Data breakpoints as implemented by source-level tools have an impact on the performance of a program during debugging because the data must be tested whenever there is a chance that its value has changed. (See Wahbe [1992] and Wahbe, Lucco and Graham [1993] for recent work that reduces the impact of data breakpoints.) Other variants of breakpoints are 1) a *conditional code breakpoint* that only triggers if a specified expression evaluates to true when a code location is reached, 2) a *temporary breakpoint* that disables itself after it triggers, and 3) a breakpoint that only triggers after a certain number of times (sometimes called a *counting breakpoint*).

5.10.1 Domain-Specific Breakpoints

The control of execution plays a vital role in most source-level debugging sessions. For domain-specific monitoring it is just as important, but we must reformulate what we mean

by "breakpoint" a little bit to fit the situation. It is inappropriate to use function names, line numbers or similar entities to specify code breakpoints in a domain-level monitoring environment. The central assumption of such an environment is that programmers do not need to know or want to know the details of the internals of their program (at least the components that they are reusing). Thus they cannot be expected to know which line number they want to stop at, for example. Similarly, when the data representation used by a component is hidden, it is impossible to express data breakpoints on that data.

So what alternatives are there? When a breakpoint triggers, it represents the attainment of some condition in the execution of the program. In our monitoring framework we use events to model program conditions. When a condition is attained we generate an event to signal that fact to monitors. Importantly, there is no other way for the frontend to determine precisely when something happens in the program. Thus we adopt a stance that breakpoints in a domain-level monitoring environment are to be stated in terms of events. For example, we might say that we want the program to stop when an event of type X is generated. Interestingly, by defining breakpoints in terms of events, we contrast with many event-based debuggers such as Dalek (Olsson et al. [1991]) that define events in terms of source-level breakpoints.

Events typically express broad conditions. Many instances of most event types will be generated by a program during its execution. Event instances have attributes to allow us to tell them apart. Fine-tuning of breakpoints based on events can be done by testing the values of event attributes. Thus we can describe breakpoint conditions that range from extremely general (any event of type X) to extremely specific (an event of type X that has exactly these attribute values). Of course, a subset of attributes may be tested or permissible values might fall into ranges, thereby providing a middle ground.

Breakpoints based on events are quite a bit different than the breakpoints we are used to in source-level debuggers. Event-based breakpoints usually have no explicit connection to the implementation of a component. Instead they are based upon the abstract model that the component presents to the outside world through its monitoring interface. For example, typical breakpoints like "stop when line n of file f is reached and variable x holds value v" might be replaced with "stop when value v is inserted into the string table". Implementation details about what happens at line n of file f and what is stored in variable x are hidden behind the monitoring interface. Assuming the presence of events with suitable meanings, event-based breakpoints unify the concepts of code breakpoints and data breakpoints.

The procedure for specifying a breakpoint can be summarized as:

1. Specify the event type that expresses the desired general breakpoint condition.

2. Write an expression using the attributes of the chosen event type, that describes the particular breakpoint desired.

3. Tell the monitoring system to cause the program to stop when an event instance of the event type from 1) is generated such that the expression from 2) evaluates to true.

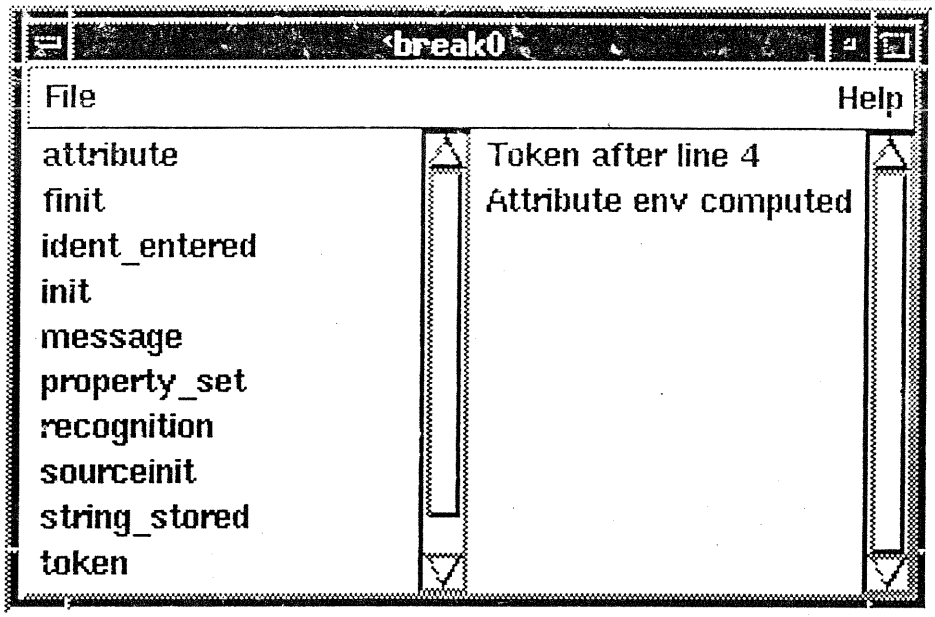


Figure 5.15: A Breakpoint Monitor.

Figure 5.15 shows a breakpoint monitor that follows this procedure. The left list gives the event types that are supported by the current subject. The right list gives the names of breakpoints that currently exist. Names are assigned by default but the defaults can be overridden by the user. The breakpoint interface can be used to create, delete, disable, and enable breakpoints.[7] In this case, the user has created breakpoints expressing the following two conditions: 1) a token on a line after line 4 has been recognized, and 2) an attribute whose name is "env" has been computed.

Breakpoints are entered or changed using a different interface. Figure 5.16 shows the user in the process of creating a new breakpoint, one that expresses "the string "begin" has been inserted into the string table". This dialog window is the result of selecting the `string_stored` event type in the main breakpoint monitor. The dialog gives information about the event type (see the interface in Section 5.3.3) and allows the user to specify the name of the breakpoint and the expression (if any) that should be used to trigger it. Expressions can access event attribute values by prefixing their names with a dollar sign.[8]

[7]Disabling has the effect of temporarily rendering a breakpoint ineffectual. It can be re-enabled at a later point in time. Disabling and enabling is typically much easier than deleting and recreating.

[8]More details on the permissible breakpoint expressions is given where the framework implementation is described in Chapter 6.

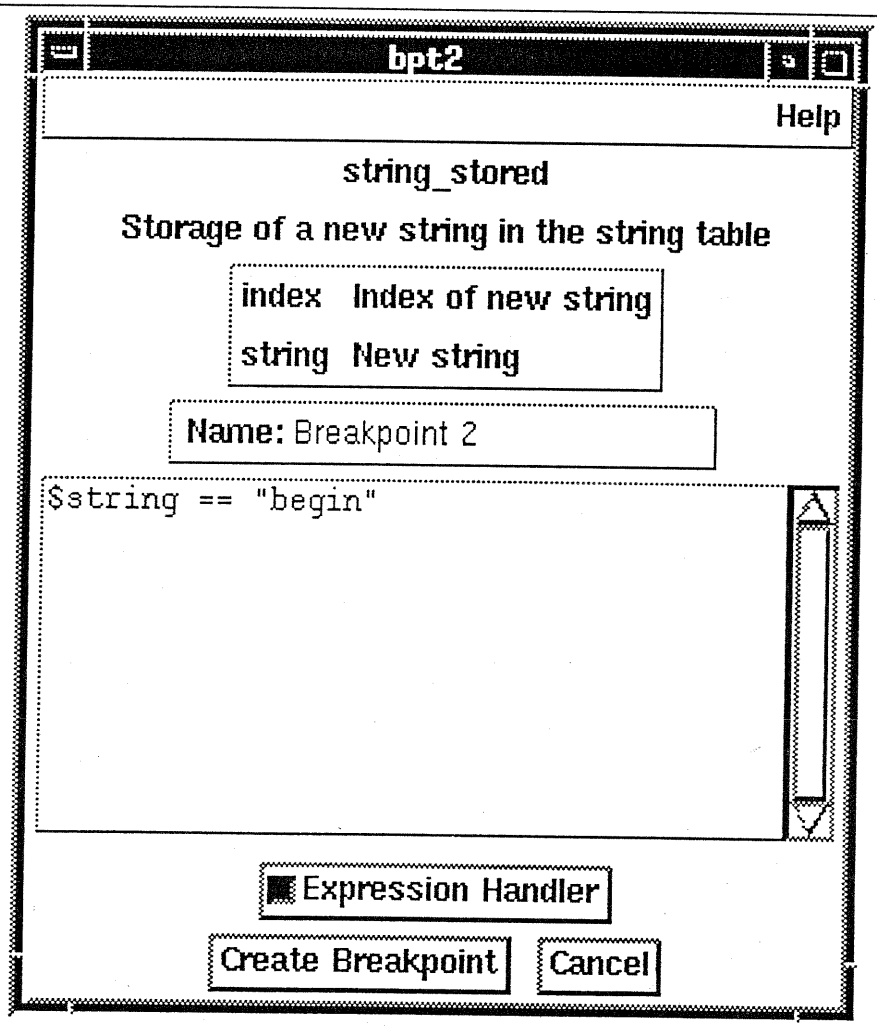


Figure 5.16: Creating a Breakpoint.

5.10.2 Monitoring Interface

Breakpoint monitors represent a bit of a departure from domain-specific monitors in their use of the subject's monitoring interface. Domain-specific monitors use the interface to obtain information about the execution and display it to the user. In this setting the actual interface events and operations are hidden from the user behind the user interface of the domain-specific monitor. However, a breakpoint monitor presents the monitoring interface directly to the user by virtue of the fact that breakpoints are expressed in terms of event types and attributes.

The desirability of having the user directly interact with the monitoring interface depends on the nature of that interface. Recall from Chapter 3 that events abstract the state changes of the program. Domain-specific monitors use this abstraction to enable them to react to changes without knowing any of the internals of the program. A breakpoint monitor allows users to set breakpoints on any of these state changes. In an appropriately designed interface, the supported abstractions form a unit that matches the user's model of the problem domain. Where this is the case, a breakpoint monitor presents event types that the user can use purposefully. For example, in Eli the lexical analysis monitor uses `token` events to present a useful interface to the generated token stream. Since Eli users have a model of the problem domain that includes the concept of tokens being generated from the input text, they can both usefully interact with the lexical analysis monitor *and* use the `token` event in a breakpoint monitor to control execution. Of course, much depends on the appropriateness of the monitoring interface, but that is true of any software system that relies on interfaces for information-hiding and abstraction.

Breakpoints are implemented using handlers that express the conditions under which the breakpoint should trigger. Recall from Section 4.2.2 that any handler that returns `TRUE` will cause execution to stop. Thus the pseudocode for a conditional breakpoint handler is :

```
if condition then
   return TRUE;
end
```

The breakpoint handler specification dialog shown in Figure 5.16 creates this form of handler by default.

If the "Expression Handler" button in Figure 5.16 is turned off, the monitor treats the specified text as the complete handler. In this case the user must arrange for the handler to return the appropriate values at the appropriate times. This facility can be used to implement more complex forms of breakpoints. For example, a handler of the following form implements a counting breakpoint that only triggers after `n` attempts:

```
if n == 0 then
   return TRUE;
else
   n = n - 1;
end
```

We assume that the language in which handlers are expressed has some mechanism for specifying data that persists between handler invocations. We also assume that `n` is initialized to an appropriate value, perhaps by a handler for the `init` predefined event type (see Section 4.2.2). More elaborate schemes can be used to get breakpoints that start counting again after they have triggered and so on. Combinations of counting and conditional breakpoints are also easy to express.

5.11 Profiling

Profiling is the production of summary information about the run-time behavior of a program (Johnson [1982]). For code segment sizes ranging from single lines to procedures, *execution profiles* present the number of times each code segment was executed (a *frequency profile*), or the amount of execution time that was expended on that segment (a *time profile*). Often the information is presented using the call graph of the program to illustrate the program's control flow (Graham, Kessler and McKusick [1983]). Profilers have also been used to obtain information on other aspects of execution such as memory allocation (Zorn and Hilfinger [1988]).

5.11.1 Domain-Specific Profiling

Conventional profilers give their output in terms of source code concepts such as function names, line numbers or variables. While appropriate in many settings, this approach is unacceptable in a domain-level monitoring environment. Our aim is to present useful information *without* requiring the programmer to be an expert on all the code they are using. Where component implementations are being reused, we must allow profiling that allows exploration of the behavior of each component from the perspective of a client of the component rather than its author.

Domain-specific events can be used to provide profiling facilities of this kind: 1) frequency profiles for different event types, and 2) time profiles for different program components. In both cases we can use component and event names to abstract program performance into the problem domain.

When a domain-specific program runs, it generates events of a variety of different types. In many cases the frequencies of those events can tell us useful things about the behavior of a program. For example, in the Eli setting one measure of the efficiency of a grammar is the number of recognitions performed by a parser for that grammar on a typical input. Different grammars may perform different numbers of recognitions. A frequency profiler can allow the programmer to compare the performance of two parsers according to this measure.

Figure 5.17 shows a monitor displaying the event frequencies observed when compiling the Pascal– GCD program. The monitor lists the frequencies for each event type. Each count is given with the components that were active when those events were generated. For example, the monitor in Figure 5.17 shows that 172 `recognition` events were caused

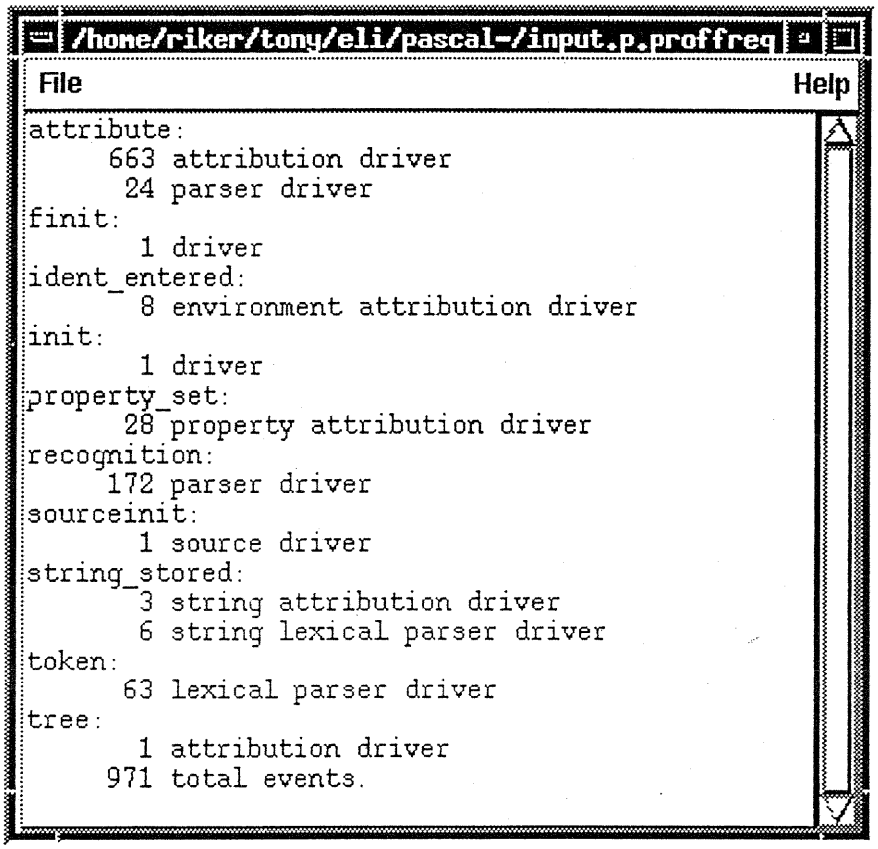


Figure 5.17: Profiling Event Frequencies.

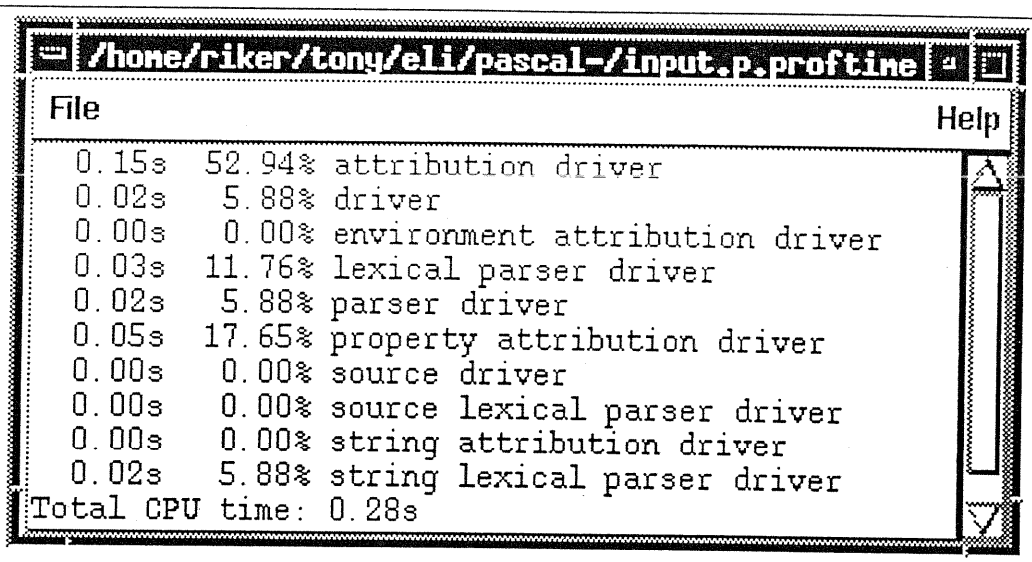


Figure 5.18: A Time Profile.

by the parser component. Sometimes more than one component name is active; the list "x y z" can be read as "component x used by component y used by component z".

We can make some specific observations from the frequency profile. For example, 663 `attribute` events were generated directly from attribution. An additional 24 `attribute` events were generated during parsing before attribution commenced. The latter correspond to the insertion of intrinsic attributes values for non-literals (computed during lexical analysis) into attribute storage in the abstract tree being constructed by the parser.

The frequency profile also shows a breakdown for strings stored in the string table. Six are stored during lexical analysis and three during attribution. The former are the strings that are used in the program: `GCD`, `X`, `Y`, `INTEGER`, `READ`, and `WRITE`. As seen in Section 5.7.2, the attribution creates an environment to hold standard identifiers. `INTEGER`, `READ`, and `WRITE` are standard identifiers but they have already been added to the table by the time attribution begins. The three strings added during attribution are the standard identifiers that have not already been added: `BOOLEAN`, `FALSE` and `TRUE`. Note that the profiling has revealed a small optimization that could be performed. For this input, there is no need to add `BOOLEAN`, `FALSE` and `TRUE` to the standard environment because no program text refers to them. Similar situations arise for most inputs.

Where frequency profiles can reveal logical errors or possible optimizations, time profiles can reveal performance bottlenecks in a program's execution. For example, if a profile shows that a compiler spends a lot of its time manipulating properties of keys, it may be appropriate to examine the use to which these properties are being put. The profile may also indicate that the implementation of the property-key component should be looked at by its author.

Figure 5.18 shows a time profile obtained when compiling the GCD program.[9] Each

line gives the amount of CPU time consumed by a particular component when used by other components as both an absolute amount in seconds and a percentage of total CPU time. For example, in this case the attribution component was responsible for more than half of the execution time of the program. We can see from the last two lines that, for this execution, the insertions of strings into the string table by the lexical analysis component were more expensive than those by the attribution component.

Usually profilers operate by themselves to give a post mortem summary of the entire execution of a program. By using our profiling monitors in conjunction with other parts of our monitoring environment we can get more fine-grained control over the information presented in profiles. All we need to do is make the profilers update their displays every time the program stops. Then it is possible to use breakpoints to obtain profiles at different points during execution. Depending on the structure of the program, this technique may reveal logical errors or performance problems that would be hidden if a profile of the whole execution was considered.

5.11.2 Monitoring Interface

Profiling monitors use monitoring interfaces in a generic fashion. They put no semantic interpretation on any of the events they observe. Both frequency and time profilers use two event types to detect when execution moves from one component to another. All components must generate an `enter` event when their code is entered and a `leave` event when it is left:

```
event enter "Enter a program component"
   (str name "Name of component");
event leave "Leave a program component"
   (str name "Name of component");
```

A time profiler simply collects the timing information from sequences of `enter` and `leave` events (see Section 4.2.3). (Recall that the event manager adjusts event timestamps to remove any monitoring overhead.) A frequency profiler observes all event generations and tabulates them under the currently active components determined from `enter` and `leave` events.

5.12 Summary

The domain-specific components described in this chapter cover a wide territory. Some are simple, self-contained components that are used as-is in many Eli-generated programs. The message, source, string table, and environment components are in this category. Others are much more complex and are generated by Eli from specifications. The lexical analysis, parsing, attribution, and property-key components are the generated

[9]Since the running time of the Pascal– compiler is short, the time profile results vary considerably from run to run. As in any profiling exercise, care must be taken when interpreting the results.

components we described. An Icon program's string scanning component is formed from 1) code generated from the operations in the program, and 2) a fixed portion of the language's run-time system.

For each component we described some desirable monitoring capabilities and designed monitors to support them. Some of the monitors are browsers: the data to which they provide access resides in the subject not the monitor. Value monitors are the simplest browsers of all because they just display simple values and invoke other monitors to browse complex data. String table monitors are also fairly simple browsers, key and environment browsers are slightly more complex, and attribution monitors are the most complex because they browse the central data structure in most Eli-generated programs: the abstract syntax tree. Other monitors work with data that does not reside in the subject throughout the execution of the program so they are forced to store it themselves. Lexical analysis monitors store the token stream, and parsing monitors store the parse tree.

Some of the domain-specific monitors illustrate special techniques that are useful in a variety of situations. Lexical analysis monitors use the predefined `stopped` event to indicate the progress of the program. This is extremely important as it allows users to correlate information presented by monitors with the program's current phase of processing. Attribution monitors communicate with value monitors by generating synthetic events, illustrating how the framework can be used for general inter-monitor communication. Attribution monitors also illustrate a situation where event translation is helpful to provide a useful monitoring capability (the examination of values of `INCLUDING` expressions). Attribute value browsers show how it is often necessary for browsers to use events to make sure they adapt to browse the correct data when the program is run multiple times.

We also presented some generic monitors: ones that use monitoring interfaces without placing semantic interpretations on the events and operations in those interfaces. We have seen that it is possible to build powerful breakpointing and profiling monitors using events. Domain-specific breakpoints unify the concepts of code and data breakpoints into one facility that matches the level of a user's understanding of their program. In conjunction with breakpoints, profiling monitors can provide fine-grained profiles in addition to complete execution profiles. Because they interact with the program at an abstract level, generic monitors provide information that is difficult to obtain with source-level tools without requiring significant knowledge of component implementations.

All of these monitors have been built using simple monitoring interfaces. The biggest interface, that for attribution, involves only three event types (counting the synthetic one), eight operations and one translation. Thus we see that it is not necessary to have wide interfaces between the monitors and the subject. Component developers can support monitoring with relatively little effort.

5.13 Related Work

As well as being illustrations of the applicability of the domain-level monitoring framework, the monitors described in this chapter provide extremely useful functionality for programmers working within their application domains. In this section we briefly discuss some previous work that has sought to provide similar functionality.

5.13.1 Compiler Monitoring

In a book based on his thesis work, Lee refers to the need for a Language Designer's Workbench that supports the "development and direct implementation of formal specifications of all aspects of programming languages" (Lee [1989, p. 127]). Eli is such a workbench. Lee also correctly observes that "debugging and testing of specifications requires special support". The present work addresses this issue specifically for compiler development within Eli.

The AGX system developed at the University of Colorado (Gray [1989]) allowed the monitoring of attribution evaluation in a previous version of Eli. Abstract tree node visits are made explicit in AGX. Breakpoints are set by designating nodes rather than attributes and are triggered when those nodes are visited by the evaluator. At a breakpoint, attribute values can be displayed. In contrast, our monitors hide the visit sequencing from the user on the grounds that it is determined automatically by an internal Eli tool and thus can change drastically due to small changes in the attribution. For example, using AGX it is impossible to know beforehand *which* visit to a node is the one during which it is appropriate to examine an attribute's value; visits that occur before the attribute value has been computed may give confusing information. Another drawback of AGX, due to it being a prototype, was that it was interfaced to the subject using *ad hoc* mechanisms rather than general principles. As demonstrated in this chapter, attribution monitoring can be supported by generally useful primitives.

The University of Washington Illustrated Compiler (Andrews, Henry and Yamamoto [1988]) provides a visualization environment that is similar to the monitoring environment described above. Displays of compiler tasks such as scanning, parsing and symbol table management can be used by students to aid their understanding of the operation of the compiler. Only one compiler has been illustrated and all illustrations were created by hand to fit the language being compiled. In contrast, our monitors build on the generation capabilities of Eli to provide monitoring for any language processor without any special hand work. Also, the focus of the Illustrated Compiler is to enhance understanding of the internals of compiler components, whereas our concentration is on monitoring the components without reference to underlying details.

Recent work for the vpo optimizer allows monitoring of the transformations applied by a program optimizer (Boyd and Whalley [1993]). The xvpodb viewer shows the register transfer instructions manipulated by the optimizer. Messages are sent from vpo to xvpodb when transformations are applied. Xvpodb allows transformations to be run

backwards or forwards. Breakpoints can be set on particular transformations or when particular instructions are affected by designated optimizations. This application illustrates the possibilities for the application of our framework to compiler development beyond the monitors described in this chapter. The framework contains general mechanisms to implement all of the functionality in xvpodb that is specially coded at present.

5.13.2 Icon Monitoring

The latest versions of the Icon system implementation contain some monitoring facilities similar to those described in this chapter (Griswold and Jeffery [1992]; Jeffery [1992a]). In particular, low-level event generation has been added to the system, intended mainly for program visualization. A monitor similar to our string scanning monitor could be created using these facilities. However, there are a number of differences: A special multi-threaded version of the Icon interpreter must be used (Jeffery [1992b]). Furthermore, monitors are written in Icon, execute in the same interpreter invocation as the program being monitored and access the event stream via a polling interface. Also, no data abstraction is provided; monitors just access program data directly.

Chapter 6

Implementation

In this chapter we describe the important facets of the *Noosa* system: an implementation of the monitoring framework of Chapter 4 for programs constructed on UNIX[1] systems using the C programming language (Kernighan and Ritchie [1978]). We cover the major design decisions made during the development of Noosa. To illustrate the impact of using Noosa for practical monitoring we also present measurements of its application to the monitor designs described in Chapter 5. Overall, Noosa has turned out to be a very flexible, moderately efficient platform for supporting execution monitoring.

6.1 Process Structure

An implementation of our monitoring framework must contain four major logical elements: the subject, the monitoring frontend, the event manager and the operation dispatcher. Communication between the subject and the frontend is mediated by the event manager (in the case of handler registration and event generation) and the operation dispatcher (in the case of operation invocation and result return). Since Noosa is intended to operate in a UNIX environment we must find appropriate physical representations in UNIX for each framework element. The central computational unit in a UNIX system is the *process*. Thus we must determine the *process structure* of our implementation: the mapping between elements and processes.

The desirable process structure is largely influenced by communication, the effect of the structure on the subject, and security. The communication design is likely to strongly influence the overall development of an execution monitoring system since information flow between the subject and the frontend is so important. Also, we should try to minimize the number and impact of changes that must be made to programs to allow them to be monitored by the system. The applicability of the system will be reduced if the necessary changes are too numerous or have too many side-effects on parts of the program that are not being monitored. Needless to say, the functional behavior of programs should not be changed by monitoring. We remove parallel programs from

[1]UNIX is a registered trademark of UNIX System Laboratories, Inc.

consideration as subjects because we can expect monitoring to effect their functional correctness. Security is important in situations where there is a possibility that incorrect program operation can affect the correct operation of the monitors themselves. All of the process structures we consider are secure in that they isolate the subject and the frontend from each other, so in the following we concentrate on the impact of process structure on communication and the subject.

Putting all the elements into a single process has the advantage that communication is relatively simple. By implementing the subject and the frontend as separate threads of control in a single process, they can execute independently while still being able to easily exchange information. A drawback of the single process approach is that it may require significant modifications to the program being debugged to transform it from a regular UNIX program to one suitable for running as a thread.

Another alternative is to use two processes: one for the subject and one for the frontend. Clearly communication is more complicated than in the single process solution because data must be transferred between address spaces. On the other hand, if network-based communication primitives are used, a dual-process structure allows the frontend to run on a different node in the network from the subject. This decoupling enhances flexibility and can increase overall performance. Running the subject in its own process also means that modifications necessary to adapt it to the monitoring system are minimized since changes are restricted to those parts of the subject that are being monitored. Other parts of the program will not have to change.

We can also imagine an organization with a single subject process and a frontend comprising a collection of monitor processes, one per monitor instance. While separation between monitors may be desirable it can complicate matters unduly. Since monitors, unlike user programs, will not need to exist outside the monitoring environment we are not motivated to adopt a design that has them executing as independent processes.

Noosa uses a dual-process structure. The frontend process incorporates implementations of all the monitors available in the system. The user interacts with this process to create monitor instances, specify monitoring actions and control execution of the subject. During a monitoring session the frontend creates subject processes in response to user requests to execute the program. At most one subject process is active at any one time.[2] The subject process runs an executable formed from the user's program augmented with monitoring support. The exact form of that support will be described in the following sections.

6.2 Communication

Communication between the two processes is at the center of the operation of the execution monitoring system. Event generation involves communication of information from the subject to the frontend. Invocation of data operations entails the frontend sending an operation description to the subject, possibly with some argument values, and the

[2]From now on the term "subject process" will refer to the currently active subject process.

subject possibly responding with a return value. We can expect the exact protocols obeyed by frontend-subject communication to vary considerably between different monitors for different program components. For example, some events will have attributes, others may not. For this reason it is desirable to have a good deal of flexibility in the communication mechanism. More precisely, it must be easy to add new message types and specify how messages of those types are to be responded to.

Noosa achieves flexibility by reducing all communication to the transfer of arbitrary text strings between the two processes. Strings represent both messages and responses. Thus the low-level communication primitives need only implement a simple string transfer protocol. No interpretation of the strings is performed at this level so new message types can be encoded as strings without affecting the communication primitives.

Building upon the string transfer mechanism, it is necessary to define the encoding of information into the strings. Rather than invent a new language and build an evaluator for it, Noosa uses the existing Tcl extension language (Ousterhout [1990]). Tcl is a full-featured imperative programming language with facilities for manipulating numbers, strings, lists, associative arrays, and files. It is specifically designed to be used as an embedded scripting language for extensible programs.

To enable the frontend and the subject to process messages, both of their processes contain complete Tcl interpreters. Each message string is treated by its receiver as a Tcl expression and evaluated. The result of the evaluation is returned as a string in response to the message. The communication mechanism can thus be seen to be just a particular form of remote procedure calling.

Using Tcl to express messages results in a great deal of power for very little implementation effort since an existing interpreter can be used. Any valid Tcl expression can be used as a message. As a trivial example, the frontend can evaluate the expression "four plus five" by sending a message containing `"expr 4+5"` to the subject which will return the answer `9` produced by the primitive Tcl procedure `expr`. More useful Tcl procedures can be invoked remotely. To define a new message type one need only define a new Tcl procedure in the receiver's interpreter. For example, sending the string `"doit 42 fred"` will cause the `doit` procedure in the receiver to be invoked with arguments `42` and `fred`. The result of the procedure invocation is returned as the message response. This technique allows complex protocols of interaction to be easily built up using Tcl instead of explicitly programmed low-level details.

Sometimes it is useful to be able to send a message without needing a response. Consequently, Noosa contains a communication primitive that operates as described above but discards the result of evaluating the message. The sender is free to continue execution immediately after sending the message.

Noosa's communication mechanism uses networking primitives, so it is not dependent on the parent-child relationship of the frontend and subject processes. Consequently, it is possible to use a source-level debugger at the same time as using Noosa. It is simply a matter of running the source-level debugger as the child of the frontend and running the subject as a child of the debugger. Domain-level monitoring can then be applied to

reusable components in conjunction with source-level debugging applied to application-specific components.

The utility of Tcl in Noosa is not restricted to the communication mechanism. Because Noosa monitors have a need to display information to the user, preferably in a windowing environment, the current Noosa monitors make use of the Tk toolkit (Ousterhout [1991]) which provides excellent graphical user interface primitives for Tcl. All monitors are coded completely in Tcl using the Tk extensions.

6.3 Data Operation Invocation

We now consider how to realize the specific functionality of the monitoring framework using the general communication mechanism. This section describes the invocation of data operations; the next three outline Noosa's event mechanism.

Data operations should allow monitors to access the data of the subject. They play a central role in limiting the amount of information that must be transmitted to the frontend because they can be invoked only when needed. Invocation of a data operation necessarily involves four steps:

1. Identification by the frontend of the operation to be invoked and any required arguments.

2. Transmission of the operation identifier and arguments to the subject.

3. Execution of the operation by the subject.

4. For an operation that has a result, transmission of the result back to the frontend.

Of course, these steps correspond exactly to the steps performed when sending a message via Noosa's communication mechanism described in the previous section. The operation identifier is the name of a Tcl procedure. The frontend sends a Tcl expression that calls this procedure with appropriate Tcl values as arguments. The subject simply evaluates the expression to execute the operation. Returning the resulting value is automatic. Hence we see that the operation dispatcher element of the framework is physically represented in Noosa by the communication mechanism.

The only piece missing from this picture is the source of the implementations for data operations, which must be supplied by the author of the monitoring interface containing the operation. Since the data that they will access is C data it is most convenient to write their implementations in C. The Tcl system design provides a simple mechanism by which new primitive procedures with C implementations can be added to the interpreter at compile time. Noosa contains the *Dapto* tool that produces suitable primitive procedure implementations from the operation implementations given in monitoring interfaces. Dapto is also used to process event specifications as described in following sections. In effect, Dapto provides interfaces between the implementation language (C) and the monitoring language (Tcl).

As an example, consider the following Dapto specification of the `get_string` operation (see also Section 5.3.3):

```
operation get_string "Look up a string given its index"
   (int index "Index of string to be looked up") : str;
{
   RETURN (string[index]);
}
```

In contrast to Chapter 5, we now also give the operation's implementation following its signature. The effect of this Dapto specification is to define a new Tcl primitive called `get_string` taking one argument. The implementation of the primitive will be constructed from the code supplied in the specification. The Eli string table component stores its strings in an array called `string`. Here we simply return the string table element given by the index argument. The `RETURN` macro is expanded into code appropriate for returning a value from a Tcl primitive. Dapto also automatically generates code to check that the number of arguments supplied to the primitive is correct.

In summary, Dapto translates data operation specifications into C code for Tcl primitives that are incorporated into the subject's interpreter. Operation invocation is the straight-forward transmission of messages invoking the primitives. Predefined operations (Section 4.3) are handled in the same way as domain-specific operations except that they have hard-wired implementations rather than Dapto-generated ones.

6.4 Event Generation

Event generation is the first step in the life of an event instance. Once generated, an event is passed to event handlers by the event manager. We describe the Noosa event manager in the next section.

Events are generated at event generation sites as a result of executing the subject. Each site is a place in the source code of the program where (conceptually) the state change represented by the event takes place. In practice an event generation site is usually reached after the program operations that cause the state change have been executed, but this is not required by the framework. The only requirement is that the values of event attributes must be available at the site in order for them to be part of the generated event instance.

At each event generation site the program must be changed to include code to generate the event and then continue normal execution. So as to limit the perturbation of the original code as much as possible, Noosa uses procedure calls to generate events. There is an event generation procedure for each event type. A call to the event generation procedure for an event type must be inserted at all sites in the program code where an event of that type should be generated. The arguments to the call are the values of the attributes for the instance being generated. For example, recall that the Eli string table component generates `string_stored` events:

```
event string_stored
      "Storage of a new string in the string table"
   (int index "Index of new string",
    str string "New string");
```

The routine responsible for storing strings in the table has the following call added to it:

```
generate_string_stored (numstr, string[numstr]);
```

where `numstr` is the variable in the string table component that holds the index of the most recently stored string. (The Dapto types `int` and `str` are represented using C integers and character pointers, respectively.)

The predefined `init` and `finit` events are handled just like domain-specific events. Dapto creates `generate_init` and `generate_finit` procedures that must be called at the appropriate moments in the execution of the subject. The former is called after the subject's Tcl interpreter has been initialized; the latter is called when the subject is about to exit.

The implementations of event generation procedures are generated by Dapto from the event signatures in the monitoring interface specifications. The procedures are designed to be able to run without Noosa present, allowing testing and monitoring to be interleaved without requiring recompilation of the program. Each generation procedure is structured so that it only generates an event if Noosa is present. Since event generation code is the only addition to the program that is executed without initiation by the frontend, the effect is that the program operates as it would if it contained no monitoring support (apart from a slight slowdown due to the ineffectual calls to generation procedures).

6.5 Event Manager

The role of the event manager in the framework is to dispatch event instances to handlers and control the execution of the subject based on the return values of those handlers. Necessary execution control amounts to stopping the program if a handler returns `TRUE` and restarting it again when appropriate. The subject is restarted by invoking the predefined `continue` operation.

The interface between the subject and the event manager is the set of event generation procedures. Each generation procedure must notify the event manager of the event instance being generated. Once the event has been handled, the event manager will notify the program whether it should resume execution immediately or stop. Recall from Chapter 4 that data operations can only be invoked while the program is stopped because that is the only situation where the frontend can accurately determine the state of the program and hence the expected outcome of operations. Thus if the program is to stop it must do so by entering a message handling loop, the implementation of which is generated by Dapto. The loop services any communication from the frontend until a `continue` operation is executed. At this point the loop is terminated and the generation procedure returns causing execution to resume.

Exactly where should the event manager be? The framework design allows the event manager to be anywhere in the system at all as long as it can interact with the subject as described in the previous paragraph. However, we can expect that typical use of Noosa will not involve all of the available event types at once. Rather, a couple of monitors may be used to diagnose a problem involving only a subset of the events generated by the subject. For this reason, it is best to locate the event manager in the subject process rather than in the frontend (or a third process) to minimize the overhead incurred when an event is generated that has no handlers registered for it.

Similar reasoning can be applied to the question of where handlers should be executed. In many monitoring situations handlers will be used to implement breakpoints as described in Section 5.10. It is reasonable to expect that the conditions for most breakpoints on an event type E will not be satisfied by most event instances of type E. Thus we want to execute handlers in the subject process to minimize the overhead for event instances that do not satisfy breakpoint conditions. An additional advantage of this scheme is that handlers can batch data in the subject before sending it to the frontend. The alternative scheme of executing handlers in the frontend forces all data to be transmitted as soon as it is generated, which may be less efficient since more individual transmissions are involved.

Handlers that do not contain conditional tests can be expected to be approximately neutral to the decision to execute handlers in the subject. In general these kinds of events just send information to monitors and usually that information is just the values of the event attributes. Hence it doesn't really matter whether they are executed in the subject or in the frontend since the amount of information transmitted is the same in either case. The only possible saving due to executing them in the frontend would be if there were many handlers for the same event type. We would save by only sending the attribute values once rather than once for each handler. Examination of the typical mode of use of the monitors in Chapter 5 indicates that this situation can be expected to occur rarely.

At this point we still have not actually specified exactly what handlers *are* in the Noosa system. The obvious choice is for handlers to be arbitrary pieces of Tcl code, since this enables the event manager to easily execute them when appropriate. Handlers are registered with the event manager using a special data operation that takes the name of the event type to which the handler is to apply and the code for the handler. The event manager keeps a list of handler code for each event type. Notifying handlers of the generation of an event instance is then just a matter of traversing the appropriate list evaluating each piece of handler code in the subject's Tcl interpreter. Handlers are conceptually registered with the event manager when they are created. However, Noosa buffers handler registrations in the frontend since it is often the case that the subject process (and hence the event manager) does not exist when monitors are being created and handlers are being registered. When the subject starts execution it requests the current handlers from the frontend and registers them.

Upon registration of a new handler, a unique *handler identifier* is returned to the frontend to represent the handler. Operations are provided to disable, enable, and remove a handler given its identifier. For example, a monitor can easily detach itself from the

subject by disabling all the handlers that it has registered. Using handler identifiers in this way also reduces the communication overhead in the common situation where handlers are temporarily not wanted such as when the user disables a breakpoint but wants to be able to get it back again without having to re-type a possibly complex breakpoint condition.

Event handlers must be able to access the attributes of event instances. As things currently stand, attribute values are passed to an event generation procedure written in C, and the handler is written in Tcl. To bridge this gap we have the generation procedure use Tcl system primitives to install the values of the attributes in the Tcl interpreter as global variables. Our convention is that the name of the attribute is used as the name of the global variable. Since only one instance can be in the process of being generated at any one time there is no possibility of a clash. If a handler wants the value to be safe for longer it must copy it explicitly.

We conclude this section with a few examples of handlers to give a concrete picture of how they are typically used. First, consider a breakpoint designed to stop when an Eli-generated program recognizes a token on line 5 of the input. Event type `token` has an attribute called `line` that is appropriate for this purpose. A handler that has the desired effect is:

```
if {$line == 5} {return 1}
```

The global variable `line` holds the attribute variable and can be referenced directly from the handler. The Tcl `return` primitive is used to return a `TRUE` value to trigger the breakpoint.[3] A default return value of `FALSE` is supplied by the event mechanism if a handler executes completely without an explicit `return`.

Here is a handler for `token` events that simply transmits the event information to a monitor:

```
nsend token_seen mon3 $type $line $col $lexeme \
   $len $val $code
```

In this code `nsend` is the Tcl procedure that provides an interface to the Noosa communication mechanism. The `token_seen` procedure is defined in the code for the lexical analysis monitor to store the attribute values for later use. The second argument is a monitor identifier that allows the frontend to differentiate between messages generated by different monitors' handlers. In this case the `mon3` monitor (an instance of the lexical analysis monitor) is the one that defined this handler, so any information generated by the handler should be given to that monitor and not to any other lexical analysis monitors. This naming convention is designed to permit more than one instance of a monitor class to coexist peacefully. The remaining arguments refer to the global variables holding the attributes of the event instance.

A final example shows how handlers can use global state to communicate with other invocations of themselves or other handlers. Global variables persist in the subject's Tcl

[3] In Tcl, zero represents `FALSE` and one represents `TRUE`.

interpreter throughout execution and, in particular, between handler invocations. Thus the following handler causes a stoppage when two end tokens are seen in a row:

```
if {$lexeme == "end"} {
   if {$seen_end} {return 1}
   set seen_end 1
} else {
   set seen_end 0
}
```

Appropriate initialization of the seen_end global variable is needed at the beginning of execution. We can do this using a handler on the init predefined event:

```
set seen_end 0
```

6.6 Event Translation

Event translations are used to allow program components to affect the generation of events by other components. It must be possible for a translation to change the values of event attributes and even suppress an event instance entirely. This means that translations must be able to interpose themselves in the event generation process between the time the event generation procedure is called and the time the event manager invokes the handlers.

Translation bodies must be supplied by the author of the component that needs to do the translation. Like data operations, the implementation of event translations should be written in C to enable them to interface easily with program data. As is the case with data operations, Noosa requires the implementation of an event translation to be specified in the monitoring interface. For example, recall that an event translation is used to map Eli attribute value computations for remote INCLUDING attributes from generated names to names that the user will recognize. In Section 5.6.3 we gave pseudo-code for the implementation of the translation. The real Dapto specification is:

```
translation rename_includings attribute
   "Translation for INCLUDINGs"
{
   if (strncmp (attr, "_IG_incl", 8) == 0) {
      extern char *incl_names[];
      attr = incl_names[atoi (attr + 8)];
   }
}
```

This code is inserted by Dapto into the body of the event generation procedure before the event manager is called.

The rename_includings translation tests the value of the attr attribute value directly through its C representation as a parameter of the generation procedure. If it

is a generated name like `_IG_incl5`, the translation extracts the numeric part (5), gets the new name by using that numeric part as an index into the `incl_names` array, and updates the value of the `attr` parameter with the new name. Event generation proceeds as normal with the new value. The `incl_names` array is automatically generated by code added to the Eli tool that expands `INCLUDING` constructs.

Translations can suppress an event by simply returning from the event generation procedure so the event manager is never called. They can also generate other events by calling the appropriate event generation procedures. Translations will be correctly applied to events generated by translations.

6.7 Time-stamps

Time-stamps must be attached to each event instance to enable events to be used to ascertain time-based behavior of a program. Monitors such as profilers use time-stamps to associate portions of the run-time to particular parts of the program. To enable accurate use of the time-stamps a monitoring system must take care to compensate for any overhead incurred while performing monitoring actions. For example, the process of generating events and invoking handlers consumes execution time. It is not appropriate to charge this time to the program, so it must be removed as part of the time-stamp computation. Similarly, adjustments must be made for time consumed while the subject is performing operation invocations for the frontend.

Noosa keeps a cumulative record of the time spent executing in the monitoring system. Each event generation procedure and operation implementation records the time on entry, and on exit adds the elapsed time since entry to the cumulative total. When events are generated, the current total is subtracted from the time the current event generation procedure was entered to give the time-stamp for the current instance.

Comparisons of the total run-time of monitored programs reported via time-stamps to the run-time reported by the Unix `time` command show that almost all of the monitoring overhead is being successfully removed.[4] The remainder is due to the time consumed by the program between actual transfer to the monitoring system and the recording of the time of that transfer. For monitoring activities with a reasonable number of event generations and operation invocations, the extra time attributed to the program is around one percent. This difference seems small enough that it will not preclude the reasonable use of the time-stamps in profiling. Also, informal comparison of profiles generated by Noosa and those generated by other profilers such as Gprof (Graham, Kessler and McKusick [1983]) and qpt (Larus [1993]) shows no observable difference when time allocated by these profilers to individual routines is manually allocated to components.

[4] All measurements were performed on a Sun SparcStation 2 running SunOS 4.1.2 with 32 megabytes of memory.

6.8 Aspects

Aspects represent segments of the monitoring interface of a program. The frontend must be able to query aspects to find out which monitors are applicable to the current program being monitored. Also, generic monitors are able to use aspects to display useful information to the user. For example, the breakpoint monitor described in the last chapter uses aspect information to display the documentation strings associated with event types and attributes.

Aspects are normally just part of the static description of the monitoring interface given to Dapto. To make aspect information available at monitoring time, Dapto must generate a representation of it that can be read by the frontend. Noosa uses a simple database represented as a Tcl script that is loaded by the frontend on startup of the system. The database is analogous to the symbolic information loaded from an executable image by a source-level debugger. A Noosa database for a program contains: 1) a list of the aspects supported by that program, and 2) a list of all the events that might be generated by the program. Each event type's attributes are also given and all the documentation strings from the Dapto specifications are preserved.

6.9 Measurements

It is useful to consider the effort that adding monitoring support to a component requires. Table 6.1 shows the sizes of each of the monitoring interfaces for the components described in Chapter 5.[5] We give the size of the Dapto specification for each component; the number is split into that due to interface specification (signatures) and that due to implementation of operations and event translations. These figures illustrate that relatively little effort is needed to specify monitoring interfaces.

Another useful measure is the number of lines of code that must be added to components to support monitoring.[6] We distinguish between code added to fixed components and code added to component generators. Analysis of the Eli components and the Icon string scanning component showed that they needed very little support for monitoring. Fixed components on the order of 100-200 lines of code needed around five percent extra code. Generators (each containing over 8000 lines of code) needed less than one per cent extra code. When the extra code needed to implement operations and translations is included (Table 6.1) the percentages increase slightly. Thus for these components the impact on code size of adding Noosa support is small. Since these components are relatively typical of reusable software used by many programmers it can be expected that the impact figures will be relatively independent of problem domain.

As mentioned at the end of Section 6.4, a program containing monitoring support can be run without Noosa present. The only differences between such an execution and an

[5] All code measurements in this section are in terms of non-blank, non-commented lines before any necessary preprocessing is done.

[6] Of course, any measurements based on lines of code must be taken with a grain of salt because of variances introduced due to formatting conventions etc.

Table 6.1: Sizes of Monitoring Interfaces in Lines.

Component	Dapto Specification		
	Signatures	Implementation	Total
Attribution	35	70	105
Environment	10	17	27
Icon String Scanning	9	0	9
Lexical	9	0	9
Message	7	0	7
Parser (COLA)	11	4	15
Parser (PGS)	11	6	17
Property-Key	13	31	44
String Table	8	19	27
Source	3	0	3
Total	116	147	263

execution of the unchanged program are: 1) the executable program contains extra execution monitoring code, and 2) event generation procedures are called at various places and immediately return. In practice, these differences result in minimal execution overhead. Noosa, as applied in Eli, brackets all monitoring code with preprocessor tests that enable that code to be omitted during compilation. Consequently, the entire monitoring overhead can be easily avoided if necessary.

6.10 Summary

This chapter has shown how the execution monitoring framework of Chapter 4 has been realized in Noosa, a practical monitoring system. The subject and frontend execute in two separate processes to minimize the changes that must be made to programs in order to monitor them. A general communication mechanism allows a wide variety of possible interaction styles between monitors and the subject. Operation invocations are simply messages sent from the frontend to the subject. Again to minimize the impact of monitoring, procedure calls are used in the program to generate events. Event generation procedures dispatch events to the event manager which executes event handlers in the context of the subject process to limit unnecessary communication. Event translations and time-stamps are supported through straight-forward additions to event generation procedure implementations.

Chapter 7

Conclusion and Future Work

Abstraction is a proven technique for reducing the complexity of software development. For example, high-level data types free the programmer from concerns of storage layout. Similarly, procedural abstraction allows a piece of code to be treated as a black box. This thesis has dealt with component-level abstraction: the collection of code and data to form a useful unit. Just like other forms of abstraction, component-level abstraction lets programmers concentrate on the functionality provided by a component and ignore details of how that functionality is implemented. Typical components might be abstract data types or classes selected from a library, or generated by an application generator from a specification of the component's desired functionality.

The abstractions used in a program have a profound effect on the software tools that are used to develop that program. In this thesis we have been concerned with execution monitoring tools: debuggers and time and frequency profilers. (Future work might consider other tools such as memory profilers, editors, and configuration management systems.) We have seen that abstractions like data types and procedures are well supported by source-level monitoring tools. For example, source-level debuggers allow data to be printed out according to its type rather than its representation, and breakpoints to be specified in terms of procedure names instead of code locations. On the other hand, we have observed that component-level abstraction is not well supported. It is frequently the case that a component-level abstraction utilized at coding time has to be violated at debugging or profiling time.

This thesis has studied the problem of providing execution monitoring tools that support component-level abstractions. Chapter 3 considered the problem from the view of individual components, determining that, according to the principle of abstraction, execution monitoring support must begin with those components. It is not desirable to have any other parts of the monitoring system embody information concerning the internal details of components, so it is up to the components themselves. Components use functional interfaces to hide their internals from clients. We introduced the concept of a monitoring interface to allow similar information-hiding for the monitoring support provided by a component.

Monitoring interfaces provide abstract access to the control flow and data of components. Following the lead of algorithm animation systems and parallel debugging methods based on behavioral abstraction, we defined control flow in terms of high-level events. Component-level events are more abstract than the events used in previous systems. In those systems the aim is to provide insights into how the code of the program works; the steps followed by the code are of prime importance. In contrast, we provide insights into the operation of the abstractions supported by the code; the actual steps performed are only of interest in that they support the abstractions. Similarly, previous systems provide direct views of program data. We have seen that abstractions of component data can be quite different from the data as represented explicitly by the program. We have introduced data operations in monitoring interfaces to implement data abstraction.

For most components the monitoring and functional interfaces are clearly related, however this thesis has not formally examined the extent of that relationship. An interesting question is the extent to which the monitoring and functional interfaces can be usefully joined into one interface. For example, usually some data operations in a monitoring interface duplicate operations in the functional interface, so they could be merged. However, other operations may be inappropriate for use by program components. More study is needed to determine a way to appropriately avoid having two separately maintained interfaces for each component.

Chapter 4 described a new execution monitoring framework that utilizes monitoring interfaces. Monitors are grouped into a frontend with which the user interacts; the frontend performs all of its interaction with the program being monitored through monitoring interfaces. It is not possible for monitors (and hence programmers) to access the internal details of any program component. This kind of interface enforcement is missing from previous execution monitoring frameworks.

Monitoring interfaces consist of data operations, events, and event translations. Monitors can invoke data operations to access or update program data. Sometimes it is useful for a programmer to be able to update data as execution progresses, perhaps to work around bugs that will be fixed in the source code later. Unfortunately, it is often the case that this data cannot be easily updated because there is no straightforward mapping from changes in an abstract data item to a physical representation. For example, updating the phrase just recognized by a parser may invalidate all sorts of state information maintained by the parser. The problem of updating abstract data needs more investigation.

Our event mechanism notifies monitor event handlers of the occurrence of time-stamped events during execution. Handlers permit arbitrary monitor actions to occur when events are generated. If desired, handlers can cause execution to stop after the generation of an event. Event translations address information-hiding when components are built in terms of other components. Translation allows components to alter event attributes, generate new events, or delete events entirely from the event stream before monitors are notified. Monitoring interfaces are structured using aspects (named groups of interface elements), enabling the frontend to configure itself based on the components present in the particular program being monitored.

Chapter 5 illustrated the monitoring framework by applying it to some typical components. For each component we described the relevant abstractions that we wanted to monitor, how a monitor for those abstractions might look from the programmer's perspective, and the monitoring interfaces necessary to support the monitor. A variety of components from the Eli compiler construction system and the Icon programming language allowed us to demonstrate the broad applicability of the framework as well as some specific techniques for handling various common situations. In addition to the domain-specific monitors, we described some generic monitors; that is, monitors that do not use the domain-specific semantics of the monitoring interfaces to do their job. The breakpoint monitor allows the programmer to specify arbitrary actions to happen on the occurrence of events (including stopping the program). The profiling monitors produce time and frequency displays on a component level, a capability not found in previous profilers.

In Chapter 6 we demonstrated that the framework can be used as the basis of a practical execution monitoring system. The Noosa system allowed us to easily build the monitor designs from Chapter 5. These monitors permit forms of monitoring that were previously extremely difficult. Noosa operates in conjunction with the Dapto processor, which processes monitoring interface specifications for inclusion into programs. The Tcl extension language is used to implement an extremely flexible communication mechanism that is used as the basis for both data operation invocation and event handling. The flexibility and extensibility achieved by basing the system on Tcl has been invaluable for constructing and testing monitors quickly. Being able to create custom forms of communication easily is extremely useful when monitoring interfaces are evolving.

In conclusion, this thesis has shown that powerful execution monitoring tools can be built for software based on reusable components without sacrificing the information-hiding properties of those components. Such tools should be provided by any programming system that supports the use of reusable components.

REFERENCES

ADAMS, E. AND MUCHNICK, S. S. [July 1986], *Dbxtool: a window-based symbolic debugger for Sun workstations*, Software — Practice and Experience 16, 7, 653–669.

AGRAWAL, H. AND SPAFFORD, E. H. [Apr. 1989], *A bibliography on debugging and backtracking*, ACM SIGSOFT Software Engineering Notes 14, 2, 49–56.

AMBLER, A. L. AND BURNETT, M. M. [Oct. 1989], *Influence of visual technology on the evolution of language environments*, Computer 22, 10, 9–22.

ANDREWS, K., HENRY, R. R. AND YAMAMOTO, W. K. [July 1988], *Design and implementation of the UW Illustrated Compiler*, Proc. Conf. on Programming Language Design and Implementation, SIGPLAN Notices 23, 7, 105–114.

ARAL, Z. AND GERTNER, I. [1988a], *Parasight: a high-level debugger/profiler architecture for shared-memory multiprocessors*, Proc. of International Conference on Supercomputing, St. Malo, France, 131–139.

———— [Sept. 1988b], *Non-intrusive and interactive profiling in Parasight*, Proc. ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems, ACM SIGPLAN Notices 23, 9, 21–30.

ARAL, Z., GERTNER, I. AND SCHAFFER, G. [May 1989], *Efficient debugging primitives for multiprocessors*, Proc. International Conference on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices 24, Special Issue, 87–95.

BAECKER, R. [Feb. 1975], *Two systems which produce animated representations of the execution of computer programs*, SIGCSE Bulletin 7, 1, 158–167.

BAECKER, R. M. [1969], *Picture-driven animation*, Proc. AFIPS Spring Joint Computer Conference 34, 273–288.

BATES, P. [Jan. 1989], *Debugging heterogeneous distributed systems using event-based models of behavior*, Proc. Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices 24, 1, 11–22.

BATES, P. C. AND WILEDEN, J. C. [Jan. 1982], *EDL: a basis for distributed system debugging tools*, Proc. Hawaii International Conference on Software and Systems Sciences, 86–93.

BATORY, D. AND O'MALLORY, S. [Oct. 1992], *The Design and Implementation of Hierarchical Software Systems with Reusable Components*, ACM Trans. on Software Engineering and Methodology 1, 4, 355–398.

BENTLEY, J. L. AND KERNIGHAN, B. W. [Aug. 1991], *A system for algorithm animation: tutorial and user manual*, AT&T Bell Laboratories, Computer Science Technical Report No.132.

BERRY, D. [June 1991], *Generating program animators from programming language semantics*, Department of Computer Science, University of Edinburgh, PhD. Thesis.

BERTOT, Y. [June 1991], *Occurrences in debugger specifications*, Proc. Conf. on Programming Language Design and Implementation, SIGPLAN Notices 26, 6, 327–337.

BIGGERSTAFF, T. J. AND PERLIS, A. J., eds. [1989], *Software reusability, Volume 1: Concepts and models*, ACM Press, New York, NY.

BOYD, M. R. AND WHALLEY, D. B. [June 1993], *Isolation and analysis of optimization errors*, Proc. Conf. on Programming Language Design and Implementation, SIGPLAN Notices 28, 6, 26–35.

BRINCH-HANSEN, P. [1985], *Brinch-Hansen on Pascal Compilers*, Prentice-Hall, Englewood Cliffs, NJ.

BROWN, M. H. [1988], *Algorithm Animation*, The MIT Press, Boston, Mass.

——— [Feb. 1992], *Zeus: a system for algorithm animation and multi-view editing*, Digital Equipment Corporation Systems Research Center, Research Report No.75.

——— [May 1988], *Exploring algorithms using Balsa-II*, Computer 21, 5, 14–36.

BROWN, M. H. AND HERSHBERGER, J. [Dec. 1992], *Color and sound in algorithm animation*, Computer 25, 12, 52–63.

BROWN, M. H. AND SEDGEWICK, R. [Jan. 1985], *Techniques for algorithm animation*, IEEE Software 2, 1, 28–39.

——— [July 1984], *A system for algorithm animation*, Computer Graphics 18, 3, 177–186.

BRUEGGE, B. AND HIBBARD, P. [Dec. 1983], *Generalized Path Expressions: a high level debugging mechanism*, The Journal of Systems and Software 3, 4, 265–276.

CAMPBELL, R. H. AND HABERMANN, A. N. [1974], *The specification of process synchronization by Path Expressions*, in Operating Systems, G. Goos and J. Hartmanis, eds., Lecture Notes in Computer Science, Springer-Verlag, 89–102.

CLEAVELAND, J. C. [July 1988], *Building application generators*, IEEE Software 5, 4, 25–33.

COHEN, J. AND CARPENTER, N. [1977], *A language for inquiring about the run-time behaviour of programs*, Software — Practice and Experience 7, 445–460.

COUTANT, C. A., GRISWOLD, R. E. AND HANSON, D. R. [Jan. 1983], *Measuring the performance and behavior of Icon programs*, IEEE Trans. on Software Engineering SE-9, 1, 93–103.

DELISLE, N. AND SCHWARTZ, M. [Jan. 1987], *A programming environment for CSP*, Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments 22, 1, 34–41.

DELISLE, N. M., MENICOSY, D. E. AND SCHWARTZ, M. D. [Apr. 1984], *Viewing a programming environment as a single tool*, Proc. ACM Symp. on Software Development Environments, 49–56.

DUCASSÉ, M. AND EMDE, A-M. [Apr. 1991], *OPIUM: a debugging environment for Prolog development and debugging research*, ACM SIGSOFT Software Engineering Notes 16, 2, 67–72.

DUISBERG, R. A'AMY [Aug. 1987], *Visual programming of program visualizations: a gestural interface for animating algorithms*, Proc. Workshop on Visual Languages, Linköping, Sweden, 55–66.

EISENSTADT, M. AND BRAYSHAW, M. [1988], *The Transparent PROLOG Machine (TPM): an execution model and graphical debugger for logic programming*, The Journal of Logic Programming 5, 277–342.

EVANS, T. G. AND DARLEY, D. L. [1966], *On-line debugging techniques: a survey*, Proc. AFIPS Fall Joint Computer Conference 29, 37–50.

FOLEY, J. D. AND MCMATH, C. F. [Mar. 1986], *Dynamic process visualization*, IEEE Computer Graphics and Applications 6, 3, 16–25.

GOLAN, M. AND HANSON, D. R. [Jan. 1993], *DUEL—a very high-level debugging language*, Proc. Winter USENIX Conference, San Diego, CA, 107–117.

GOLDBERG, A. [1984], *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass.

GOLDBERG, A. AND ROBSON, D. [1983], *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass.

GRAHAM, S. L., KESSLER, P. B. AND MCKUSICK, M. K. [1983], *An execution profiler for modular programs*, Software — Practice and Experience 13, 671–685.

GRAY, R. W. [1989], *Declarative specifications for automatically constructed compilers*, Department of Computer Science, University of Colorado, Boulder, PhD. Thesis.

GRAY, R. W., HEURING, V. P., LEVI, S. P., SLOANE, A. M. AND WAITE, W. M. [Feb. 1992], *Eli: a complete, flexible compiler construction system*, Communications of the ACM 35, 2, 121–131.

GRISWOLD, R. E. AND GRISWOLD, M. T. [1983], *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, N.J..

GRISWOLD, R. E. AND JEFFERY, C. L. [Aug. 1992], *Writing execution monitors for Icon programs*, Dept. of Computer Science, University of Arizona, Icon Project Document 192c.

GRUNWALD, D., NUTT, G. J., SLOANE, A. M., WAGNER, D. AND ZORN, B. [Jan. 1993], *A testbed for studying parallel programs and parallel execution architectures*, Proc. International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 93) 25, 1, 95–106.

GRUNWALD, D., NUTT, G., SLOANE, A., WAGNER, D., WAITE, W. AND ZORN, B. [Apr. 1991], *Execution architecture independent program tracing*, Department of Computer Science, University of Colorado, Boulder, CU-CS-525-91.

HAARSLEV, V. AND MÖLLER, R. [Oct. 1990], *A framework for visualizing object-oriented systems*, Proc. Conf. on Object-Oriented Programming: Systems, Languages and Applications, SIGPLAN Notices 25, 10, 237–244.

HANSON, D. R. [1978], *Event associations in SNOBOL4 for program debugging*, Software — Practice and Experience 8, 115–129.

HELTTULA, E., HYRSKYKARI, A. AND RÄIHÄ, K-J. [Jan. 1989], *Graphical specification of algorithm animation with ALADDIN*, Proc. 22nd Hawaii International Conference on System Sciences, 892–901.

HENRY, R. R., WHALEY, K. M. AND FORSTALL, B. [June 1990], *The University of Washington Illustrating Compiler*, Proc. Conf. on Programming Language Design and Implementation, SIGPLAN Notices 25, 6, 223–233.

HOFFMAN, D. [May 1990], *On criteria for module interfaces*, IEEE Trans. on Software Engineering 16, 5, 537–542.

HSEUSH, W. AND KAISER, G. E. [Mar. 1990], *Modeling concurrency in parallel debugging*, Proc. Symp. on Principles and Practice of Parallel Programming 25, 3, 11–20.

INGALLS, D. [1972], *The execution time profile as a programming tool*, in Compiler Optimization, 2nd. Courant Computer Society Symposium, R. Rustin, ed., Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 107–128.

ISODA, S., SHIMOMURA, T. AND ONO, Y. [May 1987], *VIPS: a visual debugger*, IEEE Software 4, 3, 8–19.

JEFFERY, C. L. [Aug. 1992a], *Eve: an Icon monitor coordinator*, Dept. of Computer Science, University of Arizona, Icon Project Document 179a.

——— [Aug. 1992b], *The MT Icon interpreter*, Dept. of Computer Science, University of Arizona, Icon Project Document 169d.

JOHNSON, M. S. [Feb. 1982], *A software debugging glossary*, ACM SIGPLAN Notices 17, 2, 53–70.

JOHNSON, R. E. AND FOOTE, B. [June/July 1988], *Designing reusable classes*, Journal of Object-Oriented Programming 1, 2, 22–35.

KELLER, R. K., CAMERON, M., TAYLOR, R. N. AND TROUP, D. B. [May 1991], *User interface development and software environments: the Chiron-1 system*, Proc. International Conference on Software Engineering, 208–218.

KERNIGHAN, B. W. AND RITCHIE, D. M. [1978], *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

KESSLER, P. [June 1990], *Fast breakpoints: design and implementation*, Proc. Conf. on Programming Language Design and Implementation, SIGPLAN Notices 25, 6, 78–84.

KISHON, A., HUDAK, P. AND CONSEL, C. [June 1991], *Monitoring semantics: a formal framework for specifying, implementing, and reasoning about execution monitors*, Proc. Conf. on Programming Language Design and Implementation, SIGPLAN Notices 26, 6, 338–352.

KISHON, A. S. [May 1992], *Theory and art of semantics-directed program execution monitoring*, Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-905.

KNUTH, D. E. [1971], *An empirical study of FORTRAN programs*, Software — Practice and Experience 1, 105–133.

KRASNER, G. E. AND POPE, S. T. [Aug./Sept. 1988], *A cookbook for using the Model-View-Controller User-Interface Paradigm in Smalltalk-80*, Journal of Object-Oriented Programming 1, 3, 26–49.

KRUEGER, C. W. [June 1992], *Software reuse*, ACM Computing Surveys 24, 2, 131–183.

KUSALIK, A. J. AND OSTER, G. M. [Feb. 1993], *Towards a generalized graphical interface for logic-programming development*, Department of Computer Science, University of Saskatchewan, TR 93-2.

LARUS, J. R. [Dec. 1990], *Abstract execution: a technique for efficiently tracing programs*, Software — Practice and Experience 20, 12, 1241–1258.

——— [May 1993], *Efficient program tracing*, Computer 26, 5, 52–61.

LAZZERINI, B. AND LOPRIORE, L. [July 1989], *Abstraction mechanisms for event control in program debugging*, IEEE Trans. on Software Engineering 15, 7, 890–901.

LEAVENWORTH, B. M. [1977], *Structured debugging using a domain specific language*, Software — Practice and Experience 7, 475–482.

LEE, P. [1989], *Realistic Compiler Generation*, Foundations of Computing, The MIT Press, Boston, Mass.

LIAO, Y. AND COHEN, D. [Nov. 1992], *A specificational approach to high level program monitoring and measuring*, IEEE Trans. on Software Engineering 18, 11, 969–978.

LINTON, M. A. [June 1990], *The evolution of Dbx*, USENIX Summer Conference, 211–220.

LONDON, R. L. AND DUISBERG, R. A. [Aug. 1985], *Animating programs using smalltalk*, Computer 18, 8, 61–71.

LOPRIORE, L. [May 1989], *A user interface specification for a program debugging and measuring environment*, Software — Practice and Experience 19, 5, 437–460.

MCDOWELL, C. E. AND HELMBOLD, D. P. [Dec. 1989], *Debugging Concurrent Programs*, ACM Computing Surveys 21, 4, 593–622.

MEYER, B. [1992], *Eiffel: the language*, Prentice Hall, New York, NY.

——— [Mar. 1987], *Reusability: The Case for Object-Oriented Design*, IEEE Software, 50–64.

MUKHERJEA, S. AND STASKO, J. [May 1993], *Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding*, Proc. IEEE International Conference on Software Engineering, Baltimore, MD.

MYERS, B. A. [July 1983], *Incense: a system for displaying data structures*, Computer Graphics 17, 3, 115–125.

——— [June 1980], *Displaying data structures for interactive debugging*, XEROX Palo Alto Research Center, CSL-80-7.

——— [Mar. 1988], *The state of the art in visual programming and program visualization*, Symposium on Visual Programming and Program Visualization, London.

NEIGHBORS, J. M. [1989], *Draco: a method for engineering reusable software systems*, in Software Reusability, Vol. I Concepts and Models, T. J. Biggerstaff and A. J. Perlis, eds., ACM Press, New York, NY, 295–319.

———[Sept. 1984], *The Draco Approach to Constructing Software from Reusable Components*, IEEE Trans. on Software Engineering SE-10, 5, 564–574.

OLSSON, R. A., CRAWFORD, R. H. AND HO, W. W. [Feb. 1991], *A dataflow approach to event-based debugging*, Software — Practice and Experience 21, 2, 209–229.

OLSSON, R. A., CRAWFORD, R. H., HO, W. W. AND WEE, C. E. [May 1991], *Sequential debugging at a high level of abstraction*, IEEE Software 8, 3, 27–36.

OUSTERHOUT, J. K. [1990], *Tcl: an embeddable command language*, Winter USENIX Conference.

———[1991], *An X11 toolkit based on the Tcl language*, Winter USENIX Conference.

PLATTNER, B. AND NIEVERGELT, J. [Nov. 1981], *Monitoring program execution: a survey*, Computer 14, 11, 76–93.

PONAMGI, M. K., HSEUSH, W. AND KAISER, G. E. [May 1991], *Debugging multithreaded programs with MPD*, IEEE Software 8, 3, 37–43.

POWELL, M. L. AND LINTON, M. A. [1983], *A database model of debugging*, The Journal of Systems and Software 3, 295–300.

PRIETO-DIAZ, R. AND NEIGHBORS, J. M. [Nov. 1986], *Module interconnection languages*, Journal of Systems and Software 6, 4, 117–143.

REISS, S. P. [1985], *Pecan: program development systems that support multiple views*, IEEE Trans. on Software Engineering SE-11, 3, 276–285.

———[July 1990], *Connecting tools using message passing in the Field environment*, IEEE Software 7, 4, 57–66.

ROSENBLUM, D. S. [May 1991], *Specifying concurrent systems with TSL*, IEEE Software 8, 3, 52–61.

SCHWARTZ, J. T. [1970], *An overview of bugs*, in Debugging Techniques in Large Systems, R. Rustin, ed., Courant Computer Science Symposium 1, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1–16.

SHIMOMURA, T. AND ISODA, S. [May 1991], *Linked-list visualization for debugging*, IEEE Software 8, 3, 44–51.

DA SILVA, F. Q. B. [1992], *Correctness proofs of compilers and debuggers: an approach based on structural operational semantics*, Department of Computer Science, University of Edinburgh, Ph.D. Thesis.

SNODGRASS, R. [May 1988], *A relational approach to monitoring complex systems*, ACM Trans. on Computer Systems 6, 2, 157–196.

SOSIČ, R. [July 1992], *Dynascope: a tool for program directing*, Proc. Conf. on Programming Language Design and Implementation, SIGPLAN Notices 27, 7, 12–21.

STALLMAN, R. M. AND PESCH, R. H. [Oct. 1992], *The GNU Source-Level Debugger*, Free Software Foundation, Edition 4.06 for GDB version 4.7.

STASKO, J. T. [1990a], *A practical animation language for software development*, Proc. International Conference on Computer Languages, 1–10.

——— [Sept. 1990b], *Tango: a framework and system for algorithm animation*, Computer 23, 9, 27–39.

STASKO, J. T. AND PATTERSON, C. [Sept. 1992], *Understanding and characterizing software visualization systems*, Proc. IEEE Workshop on Visual Languages, Seattle, WA, 3–10.

STEFIK, M. J., BOBROW, D. G. AND KAHN, K. M. [Jan. 1988], *Integrating access-oriented programming into a multiparadigm environment*, IEEE Software 4, 1, 10–18.

TOLMACH, A. P. AND APPEL, A. W. [June 1990], *Debugging standard ML without reverse engineering*, ACM Lisp and Functional Programming Conference.

UTTER, S. AND PANCAKE, C. M. [Jan. 1991], *A bibliography of parallel debuggers*, ACM SIGPLAN Notices 26, 1, 21–37.

WAHBE, R. [Sept. 1992], *Efficient data breakpoints*, Proc. International Conference on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices 27, 9, 200–212.

WAHBE, R., LUCCO, S. AND GRAHAM, S. L. [**June 1993**], *Practical data breakpoints: design and implementation*, Proc. Conf. on Programming Language Design and Implementation, SIGPLAN Notices 28, 6, 1–12.

WAITE, W. M. [**Jan. 1993**], *A complete specification of a simple compiler*, Department of Computer Science, University of Colorado, Boulder, CU-CS-638-93.

WALKER, K. [**Aug. 1991**], *The implementation of an optimizing compiler for Icon*, Department of Computer Science, University of Arizona, TR-91-16.

WALKER, K. AND GRISWOLD, R. E. [**Aug. 1992**], *An optimizing compiler for the Icon programming language*, Software — Practice and Experience 22, 8, 637–657.

WIRFS-BROCK, A. AND WILKERSON, B. [**May/June 1989**], *Variables Limit Reusability*, Journal of Object-Oriented Programming 2, 1, 34–40.

YOUNG, M., TAYLOR, R. N. AND TROUP, D. B. [**June 1988**], *Software environment architectures and user interface facilities*, IEEE Trans. on Software Engineering 14, 6, 697–708.

ZORN, B. AND HILFINGER, P. [**June 1988**], *A memory allocation profiler for C and Lisp programs*, Proc. Summer 1988 USENIX Conference, San Francisco, CA, 223–237.

Appendix A

The Dapto Language

This appendix contains a brief informal description of the Dapto monitoring interface specification language. The only formal notation used is a form of Backus-Naur Form to describe the concrete syntax. Rules are of the form:

lhs: *rhs*.

where *lhs* is a single symbol and *rhs* is a possibly empty sequence of symbols. Literal symbols are delimited by single quotes, alternation is indicated by a slash and the symbol `empty` represents an empty alternative. Defining occurrences of identifiers are indicated with the `iddef` symbol and identifier uses with the `iduse` symbol. Strings are indicated by the `string` symbol standing for arbitrary sequences of characters delimited by double quotes (C language escape sequences are accepted).

A.1 Monitoring Interface Specifications

```
spec:     aspects.
aspects:  aspect / aspects aspect.
aspect:   'aspect' iddef ';' elements 'end' ';'.
elements: element / elements element.
```

A Dapto specification consists of a non-empty collection of *aspects* each named with a unique identifier. Each aspect incorporates a non-empty collection of *elements*. The identifiers defined by signatures within a single Dapto specification must be unique.

```
element:     includefile / eventsig /
             operationspec / translationspec.
includefile: string.
```

Each element defines an *include file name*, an *event*, an *operation*, or a *translation*. An include file is given by a string containing its name. The named file will be included by the output generated by the Dapto processor. Event signatures, operation specifications and translation specifications define the actual elements of the monitoring interface.

A.2 Event Signatures

```
eventsig: 'event' iddef export string '(' optattrs ')' ';'.
```

An event signature defines the identifier of an event type, whether it is to be exported, its documentation string and its attributes (if any).

```
export: empty / '*'.
```

An event export marker is used to cause a flag to be set for the event in the monitoring database generated by Dapto. Conventionally the flag is used by generic monitors. Only events marked for export are made visible to the user of the monitor.

```
optattrs: empty / attrs.
attrs:    attr / attrs ',' attr.
attr:     iduse iddef string.
```

For each event attribute a type identifier (`iduse`), attribute identifier (`iddef`) and documentation string must be given. Currently the type identifier must refer to one of the two predefined types: `int` or `str`. Attribute identifiers must be unique within a single event type.

A.3 Operation Specifications

```
operationspec:
  'operation' iddef string '(' optparams ')'
    '{' code '}' /
  'operation' iddef string '(' optparams ')' ':' iduse
    '{' code '}' .
```

An operation specification defines the identifier of an operation, its documentation string, its parameters (if any) and the code implementing the operation. An alternative form of specification includes a return type (`int` or `str`) for operations that are to return values. The operation implementation is arbitrary C code that may refer to parameter values by name. Parameters of type `int` are represented as C integers; those of type `str` are represented as character pointers.

```
optparams: empty / params.
params:    param / params ',' param.
param:     iduse iddef string.
```

For each operation parameter a type identifier (`iduse`), parameter identifier (`iddef`) and documentation string must be given. Currently the type identifier must refer to one of the two predefined types: `int` or `str`. Parameter identifiers must be unique within a single operation.

A.4 Translation Specifications

```
translationspec: 'translation' iddef iduse string
   '{' code '}'.
```

A translation specification introduces a named translation (`iddef`) for an event type (`iduse`) along with its documentation string. The event type must be defined by an event signature elsewhere. The code portion of a translation specification is arbitrary C code that may refer to the attributes of the event type by name. Attributes of type `int` are represented as C integers; those of type `str` are represented as character pointers.