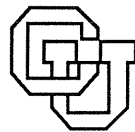
Extending Object Interfaces for Debugging

Anthony M. Sloane

CU-CS-644-93 February 1993



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Extending object interfaces for debugging

Anthony M. Sloane

CU-CS-644-93 February 1993

University of Colorado at Boulder

Technical Report CU-CS-644-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Extending object interfaces for debugging

Anthony M. Sloane

February 1993

Abstract

Object-oriented programming relies on building programs out of objects interacting through well-defined interfaces, so it is especially important that support tools fully accommodate this view. When it comes to debugging, object interfaces are often insufficient and it is necessary to violate encapsulations to discover enough to diagnose a problem. We show that extending the interfaces of objects to include extra methods and event generation enables encapsulations to be preserved. A framework is presented that allows environment builders to utilize these extensions to provide a variety of powerful debugging operations. The presence of inheritance motivates the inclusion of filtering mechanisms for events. We describe preliminary experiences with an implementation of the framework.

1 Introduction

Object technology encourages programmers to construct systems consisting of interacting objects. Each object *encapsulates* its data (state) and code (behaviour) behind a *functional interface.* Methods in a functional interface implement abstractions of the state and behaviour of the associated object.

To adequately support object technology, it is necessary to consider the impact that it has on each phase of the software lifecycle. Previous research has addressed the design phase, in particular domain analysis and object-oriented design (see [24] and [11] for overviews). Other work has looked at supporting other parts of the development cycle such as testing [15,19].

Unfortunately, existing techniques and tools do not provide very good support for object technology when execution monitoring [16] is needed. In this paper we consider the de-

bugging phase. A common property of existing object-oriented debuggers [5,12,21] is that they do not support encapsulations during debugging. Any encapsulation boundaries that exist in a program are non-existent during debugging. There is a good reason for this: object interfaces frequently do not support debugging well so it is necessary for programmers to violate encapsulations in order to be able to diagnose bugs. We argue in this paper that to fully support object-oriented development, programming environments have a responsibility to support encapsulations. This entails special work on the part of object designers to incorporate debugging support.

Consider a programmer debugging an object. He or she brings a certain amount of information along to a debugging session. There is information about the object in question: what it's supposed to do and how it's supposed to do it. There is also information about any other objects with which this object interacts. In accordance with the principle of encapsulation the programmer may know what another object is supposed to do but not how it does it. This will become more common as libraries of reusable objects become more widespread.

Suppose further that one of the server objects in question implements a string storage object with the following interface:

```
store_string (string) : int;
retrieve_string (int) : string;
```

`Store_string` returns a unique integer for each unique string that it is presented. `Retrieve_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 object. Strings can be stored and retrieved as desired. However, more may be needed when debugging an object that uses the string storage object. For example, suppose the programmer at some point wants to know the current contents of the string storage. As is, the interface does not support such a method so it is not possible to invoke it from a source-level debugger as one could invoke

`retrieve_string`, for example. One alternative is to set tracepoints on each `store_string` method. This is not satisfactory either because assembling the output from the tracepoints into a coherent view of the storage can be time-consuming and difficult.

The programmer in this case is forced to break through the encapsulation of the string storage object and examine its implementation. By looking at the code (assuming it's available) and the data of the object it may be possible to get the information needed. Of course, breaking 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 is especially true when diagnosing a bug that involves the use of the string storage module and not its operation *per se.*

Clearly the string storage module does not provide an adequate interface. In this case it suffices to add a print method (or perhaps inherit a standard print method from a more general class). A specific print format, however, is unlikely to satisfy all programmers equally. The reason is that a specific method is overly constrained and hence affects its reusability. Thus we need a general facility to access the data independent of any particular format.

Our approach advocates the development of interfaces that support general access to the abstractions supported by objects. We should expect nothing less from an object-oriented system. Conventional object-oriented development concentrates on the functionality required by other objects when designing interfaces. We argue that debugging functionality must also be taken into consideration.

Adding extra methods may not be enough to support full debugging. Section 2 uses breakpoints to motivate the inclusion of mechanisms to allow the generation of object-specific events at run-time. Section 3 shows how a general debugging framework can be built using method and event extensions. In particular, we describe how a powerful breakpoint facility can be implemented using events and show how inheritance motivates the inclusion of a filtering mechanism for events. Section 4 presents preliminary experiences with an implementation of the framework in the context of a compiler-generation system. We conclude with a discussion of issues arising from the framework design and consideration of related research.

2 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.

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

Consider an object that encapsulates a generalized parsing scheme. It has a single method 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 the 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 object supports. However, as it is, the interface 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 object. Depending on the complexity of the object implementation this may be extremely difficult.

A better way to handle this case is to extend the interface of the parser object to fully support the abstraction. In general we can do this by introducing *events* that represent abstract points during the lifetime of the object; 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 occurrences 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

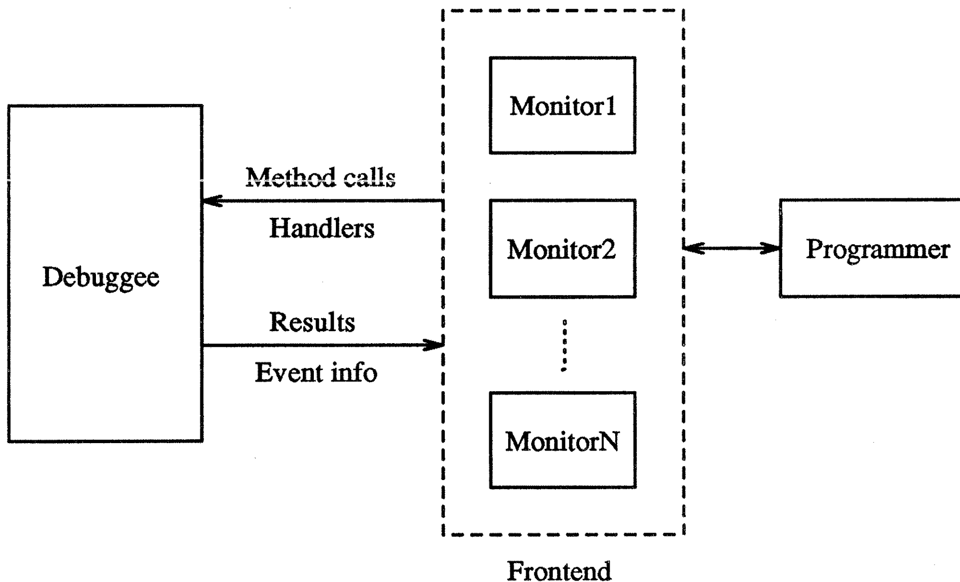


Figure 1: A general debugging framework.

production used. Determining appropriate event types and attributes is thus part of the job of designing object interfaces.

Events lie at the heart of the general debugging framework we present next. Not only do they allow abstract breakpoints to be implemented easily, they can also be used to make arbitrary information available at appropriate points during execution.

3 A General Framework

Extended interfaces (events and new methods) allow objects to be integrated into a general framework for debugging. The framework consists of a program being debugged or *debuggee* and a set of *execution monitors* grouped into a *frontend* (Figure 1). The programmer selects monitors that are appropriate for the current debugging task and interacts with them to specify debugging operations. The monitors in turn interact with the debuggee during execution to implement those operations. For this discussion we treat the debuggee as one operating system process and the monitors as (collectively) another. There is nothing in the framework, however, that prevents monitors from running in the same process as the debuggee or as individual separate processes perhaps on different machines than the debuggee.

Debuggee methods can be invoked by monitors via message passing between the frontend and the debuggee. The results (if any) are returned directly to them. As mentioned above, methods should be representation-independent. Monitors can retrieve data and display it in any appropriate form. Different monitors can display different views of the same data without interfering.

Monitors can *register* event handlers based on event type by sending them to the debuggee process. When an event of type T is generated all handlers associated with type T are called; execution of the handler takes place in the debuggee process but message passing can be used to send event information to the frontend. Arguments are passed to each handler describing the attributes of the event occurrence. The semantics of each event type and its attributes (if any) are completely specified by the objects present in the program being debugged. No particular semantic content is needed by the framework. Monitors may be freely created and destroyed by the programmer during a debugging session so it must also be possible to deregister previously registered handlers.

By registering handlers, monitors can react in any way to event generation. Handlers can perform arbitrary actions such as: indicating program progress through input data, updating a view of a program data structure, or stopping program execution because a condition is satisfied (see below).

Execution Control

The framework provides mechanisms to allow the frontend to create a debuggee process and establish the communication link between the two processes. When a new debuggee is created any current event handlers (representing currently active monitors) are registered. Notification is sent to the frontend when the debuggee exits.

To allow control over the execution of the debuggee, handlers return a boolean continuation flag. Once all the handlers for an event instance have been called, if any of them returned True, program execution stops at that point. If all of them returned False, program execution continues as normal. When execution stops, a special `stopped` event is generated. It

can also have handlers so that a monitor can notice when execution stops even when another monitor caused it to do so. When the debuggee stops it enters a wait for incoming messages from the frontend. Some of the messages will represent method calls; others are framework operations such as new handlers or an indication that execution should be resumed.

Breakpoints are easily implemented using handlers that return True when the breakpoint should be triggered. Conditional breakpoints can be implemented efficiently because handlers execute in the debuggee. This is in contrast to conventional debuggers such as GDB [18,21] that evaluate breakpoint conditions in the debugger itself thereby incurring overhead due to a transfer of control even when the breakpoint does not trigger.

Data breakpoints require the detection of a change in a specified piece of data to trigger a stop in execution. Source-level debuggers often implement data breakpoints by single-stepping the program and testing the data after each step. Recently Wahbe has shown that by patching the program with testing code, data breakpoints can be implemented relatively efficiently [23]. However, a debugger must still determine where to patch the program. For many languages it is hard to limit the number of patch locations because data could potentially change in many places. In practice most pieces of data are actually changed at few places so our framework improves the situation. Having breakpoints based on well-defined interfaces to data within objects means that the number of handlers that have to be defined is lower than the number of patches required by a conventional debugger. Thus execution is not impacted as much by redundant tests.

Aspects

Aspects are named portions of object interfaces. They collect a related set of methods and events under a common identifier and are specified as part of the definition of objects. A monitor that uses a particular aspect can be used with any object that supports that aspect.

The framework provides facilities to allow the frontend to find out which aspects are provided by a debuggee. The breakpoint monitor of the previous section relies on general capabilities of the framework. Most other monitors will provide views or operations tailored

to the types of objects being debugged. They can use aspects to query the capabilities of the debuggee. Thus an environment can configure itself to the debuggee in question. Note that this capability is not needed with conventional debuggers because all debugging operations work at the source-code level which is supported by all programs by definition.

Multiple Instances

Objects are run-time entities, they are described at coding time with static notations such as class definitions. In our framework these definitions prescribe the debugging interface to each object belonging to the class; i.e. contain the aspects of instances. Most object-oriented systems will contain more than one instance of at least some classes. The framework must allow monitors to deal with this.

If the debuggee contains more than one instance of a class it must be possible for the programmer to designate which instances are of interest during a debugging session. A browsing capability similar to that provided in Smalltalk-80 [5] is appropriate for examining objects at the source-level. An open browser can allow the creation of monitors associated with the object being browsed. Each monitor will only pay attention to events generated by the object to which it is associated.

Debugging methods already require the specification of an object to which to apply the method so no new mechanism is required to support multiple instances. In order to dispatch events appropriately, monitors must specify the object to which handlers apply at the time the handlers are registered.

Inheritance and Filtering

Inheritance allows new classes to be described as variations of existing classes. Debugging interfaces are inherited just as functional interfaces are. In some contexts debugging methods or events can be reused by a subclass without alteration. In other cases it is desirable to change some methods, some events or both. Of course, the subclass can define any number of new methods or events.

The need to redefine parts of the debugging interface arises in much the same circumstances as the need to redefine parts of the functional interface. It is important that a subclass present an interface appropriate to its function; it should not reveal any use of superclass facilities.

Consider a class that implements a deque with methods for addition and removal of elements at both the left and the right ends. Assume that events are generated to indicate the occurrence of these insertions (e.g. `add_left`, `remove_right` and so on). It is possible to use the deque class to implement a regular queue by removing the addition-at-left and removal-at-right methods and using the remaining two methods to implement the queue methods. Since we are using the deque class methods we will get deque events generated along with events for the queue (such as `add` or `remove`). However, this makes the implementation of the queue visible during debugging which we are trying to avoid. A programmer should not be able to tell or be required to know that the queue is implemented using a deque.

Dependences such as between the deque and queue classes can be hidden during debugging by allowing classes to specify *event filters*. In this case, rather than having the queue methods generate queue events, the queue class would specify a filter to turn deque events generated by queue objects into appropriate queue events. For example, `add_right` would be filtered and replaced with `add`. This mechanism places the responsibility for filtering in the only place that should be expected to know that filtering is necessary: a class that is reusing another classes methods. Thus implementation decisions are not visible to any other part of the system.

Summary

In summary, the framework allows programs consisting of collections of objects to be run under the control of a monitoring frontend. Monitors can query and observe objects using debugging methods and events. Handlers associated with object instances allow arbitrary actions, including stopping program execution, to be performed when events occur. The general breakpoint monitor uses events without any semantic interpretation so it can be used with any

program. Aspects allow domain-specific monitors to find out whether they can be applied to the current program. Event filters allow classes to translate events that are generated by their superclasses enabling inherited encapsulations to be preserved.

4 Experiences

To enable experimentation with the framework an implementation called *Noosa* has been constructed. It is currently used in the context of the Eli compiler construction system [6] With the exception of the Eli-specific monitors described below, the system is domain-independent and we intend apply it to other domains such as the Icon programming language (using monitoring for high-level constructs such as string scanning).

The objects comprising an Eli-generated program correspond to the major components of a compiler such as the lexical analyzer, parser, semantic analyzer, and auxiliary data structures like string storage and environments handling visibility of names due to scoping. Eli allows these objects to be automatically derived from high-level declarative specifications of their function. Each object is actually implemented in C, but writers are not required to have any detailed knowledge of how any object performs its function in order to write their specifications. Eli supports a particular model of compiler construction and each object encapsulates all other state and behaviour behind a well-defined interface.

Before Noosa, debugging in Eli relied on source-level debuggers for C such as GDB [21] or Dbx [10]. Compiler writers were often required to understand the operation of various objects in a significant amount of detail. Some objects are relatively simple (e.g. the string storage object) so the burden is not great. However, other objects are produced from specifications by complex processes. For example, understanding the mapping performed by Eli for the semantic analysis object is extremely difficult because it relies on a complicated translation of an attribute grammar specification.

Noosa is written in C, Tcl [13] and Tk [14] and currently has been tested on a Sun SparcStation 2 running SunOS 4.1.2. The entire system comprises 1000 lines of C for the

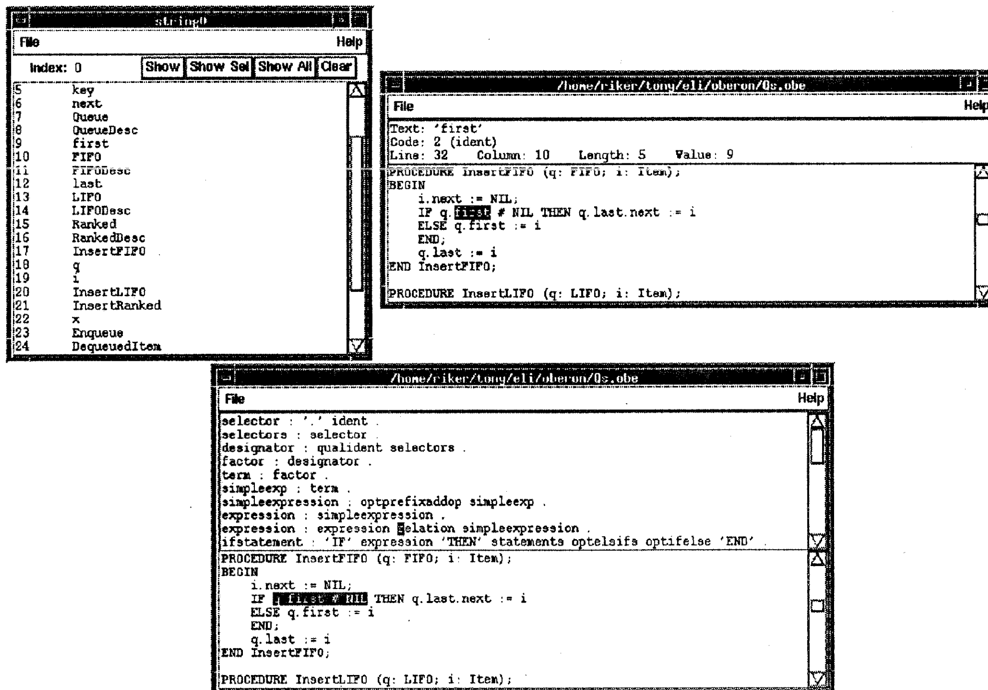


Figure 2: Eli string, lexical and parsing monitors. In the lexical monitor (upper right) the user has selected the highlighted `first` token. In the parsing monitor (bottom) the user has selected the same token and then the production corresponding to the highlighted relational expression.

framework and 1700 lines of Tcl for the frontend. About 1000 lines of the frontend Tcl is independent of Eli and deals mainly with control of the debuggee process and breakpoints. Both the frontend and the debuggee run Tcl interpreters, and monitors dynamically construct event handlers in Tcl. A simple custom-built communications library allows Tcl expressions to be sent from one process to the other and executed.[1] Both handler loading and messages from handlers to the frontend are implemented this way. Aspects are implemented using a table produced by Eli when it generates a program.

Noosa currently provides the following specialized monitors (Figure 2):

- String storage (upper left window): Allows correspondences between strings and their integer representations to be examined. Uses a `string_stored` event to provide dynamic update. Strings are fetched using an existing `get_string` method or a new `get_all_strings`

[1] The Tk `send` primitive is not used because it routes all communication through the X server making its performance unacceptable when many messages must be sent.

method. Correspondences can be changed by the user when debugging and are effected by a new set_string method.

- Lexical analysis (upper right window): Displays the current input file being used by the translator (debuggee) and highlights the current token during execution. When execution is stopped, selecting a source location with the mouse highlights the token at that location and produces a detailed description including the terminal symbol that represents that token, coordinate information and intrinsic value (e.g. string storage index for identifiers). This monitor uses token events whose attributes are the detailed token information and the stopped event at which point highlighting is performed. An added method provides a mapping between internal token codes and grammar terminal symbols.

- Parsing (bottom window): Provides a different view on the translator input file. Selecting a source location with the mouse produces a list of the productions (if any) that have been used to derive that source location. Selecting a production in that list highlights the region of source text that is derived by the production. Each recognition of a phrase by a parser generates a recognition event. An added method provides a mapping between integer production codes and the text of the productions.

Each of the Eli-specific monitors is independent of the corresponding pieces of Eli-generated programs. Thus, for example, it is possible to replace the lexical analyzer generator or parser generator used by Eli without changing the respective monitors at all as long as the debugging interfaces are not changed. In fact, Eli incorporates two parser generators; Noosa's parsing monitor can be used with either of them without change.

Implementing the debugging interfaces to support these monitors involved the addition of small amounts of code to the tools that generate the corresponding program objects. The additions are on the order of 10-20 lines of C code for each monitor representing a tiny fraction of the code of the respective tools.

The current domain-independent monitors are shown in Figure 3. The run control monitor (upper left window) provides monitor creation facilities and overall control of the

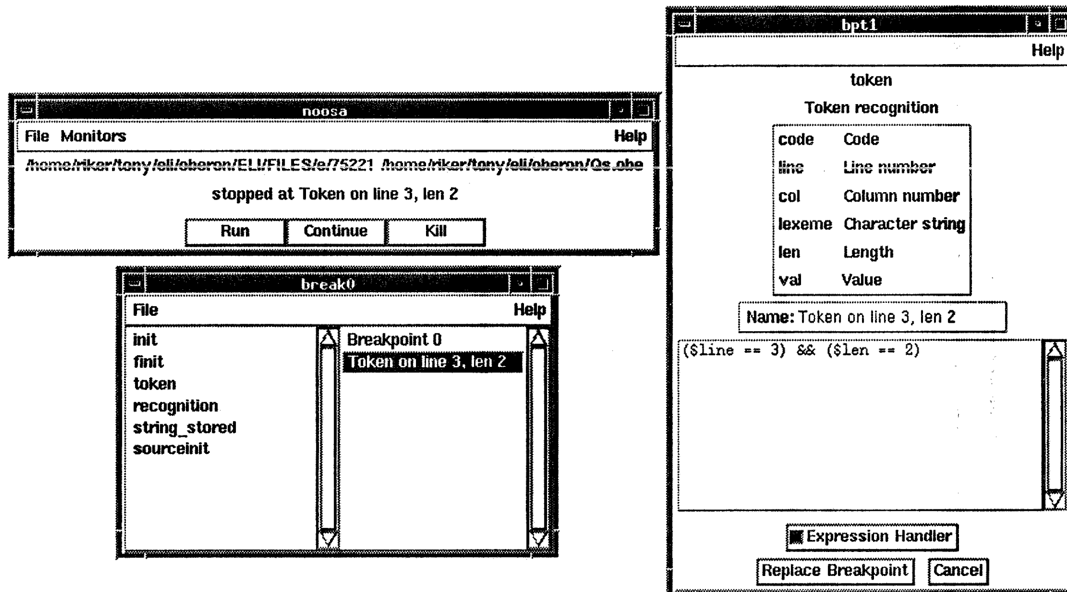


Figure 3: Run control and breakpoint monitors. The debuggee is currently stopped at the breakpoint shown in the right window.

execution of the debuggee. It uses the framework to control the debuggee process and the `stopped` event which has one attribute describing the reason for the stop (e.g. the name of the breakpoint(s) that caused the stop).

The breakpoint monitor (lower left window) allows breakpoints to be set using any of the event types listed on the left of its window. Breakpoints already created are listed by name on the right. Conditions can be specified as expressions using the event attributes listed in the breakpoint window (right) or as arbitrary Tcl code. Among other things, the latter option allows the use of global variables to propagate state between breakpoints. Thus it is possible to set a condition that tests for the prior occurrence of another event without requiring transfer of control to the frontend.

Monitors are also integrated so that, for example, breakpoints can be set by indicating entities in the appropriate domain-specific monitors. For example, only a few mouse clicks are required to set a breakpoint to trigger when a particular coordinate in the compiler's input file is recognized by the parser.

The current version of Noosa exhibits reasonable performance when small to medium-sized inputs are presented to Eli-generated programs with one of each monitor present. We are currently conducting measurements aimed at improving its performance on larger inputs (i.e. when larger numbers of events are generated).

The existing Noosa monitors do not use event filtering, but future monitors will. Inheritance *per se* is not used by Eli but an analogous situation exists if an object in an Eli-generated compiler is implemented in terms of another more general object. For example, a tool may translate a user specification into a specification for another tool. The implementation may generate events appropriate to the abstractions present in the intermediate specification which are unknown to the programmer. Filtering enables events to be translated back into the abstractions appropriate for the specifications originally supplied. A similar situation exists in Eli's library. The library provides high-level characterizations of typical sub-problems that arise during semantic analysis in a compiler. These characterizations are implemented in terms of other parts of Eli. Filters can be used to remove the requirement that programmers know how each library module they use is implemented.

5 Discussion

When designing debugging interfaces there are tradeoffs involved in deciding between the use of a new method or an event. A method allows a monitor complete access to data when and if it is needed; an event typically provides small pieces of data but provides them dynamically to permit animation, for example. In some cases it is possible to use either a method or an event. Consider the browsing of a data structure implemented by an object. The object can provide a method to access the data structure or generate events to indicate changes to it. A monitor can either use the method at the appropriate time or handle every change event and effectively keep its own copy of the data structure. This example illustrates a simple decision procedure that works well in practice to avoid duplication of data: if the debuggee maintains a data structure that is visible (i.e. is part of an object's interface) then methods should be available to access

it. Events should be generated for each change so that animations or the like can be supported. If the debuggee generates information but doesn't store it directly (e.g. recognitions of tokens by a lexical analyzer or phrases by a parser) then only events are required and each monitor should maintain its own representation tailored to its needs.

Having objects support debugging complicates their implementation. Where objects already exist it may or may not be easy to add suitable support for debugging depending on the internal details of the object. The aim of this support is to allow debugging of programs that use these objects rather than debugging the objects themselves. Where efficiency is a concern, it is appropriate to have versions of objects designed for debugging separate from production objects. Our experience with Eli has shown that even for complex objects it is usually not necessary to rewrite existing objects to get good monitoring support. It is necessary, however, to provide some way to produce a production version of a program without monitoring support, analogous to compiling a program without regular compiler debugging information.

When objects are designed from scratch, our approach advocates attention to debugging interfaces from the start. Many programmers currently spend time creating scaffolding to support debugging of their code. We simplify this process by providing a general framework that allows these programmers to concentrate on the characteristics of their objects rather than on the scaffolding. While our discussion in this paper has centered on providing debugging support that operates at the level of object interfaces and preserves encapsulation, it is clearly also possible to use our mechanisms to provide support for object implementations.

Event handlers imply the presence of a facility for dynamically loading code into a running process. Dynamic loaders such as dld [8] are suitable; alternatively, as is done in Noosa, an interpreted extension language can be used. The former approach achieves better runtime performance but loading is slower in general because handlers must be compiled before loading. Pre-compiled handlers can be used but this restricts the dynamic variability of monitors. An extension language makes loading new handlers fast but their execution is slower.

Object-level debugging as discussed in this paper does not cover all possible debugging situations. In some cases it may be desirable to use existing debugging tools in conjunction

with monitors. Eli, for example, allows C modules to be supplied by the programmer in conjunction with their high-level specifications. These C modules must often be debugged at the source-code level. Our framework easily allows this when a flexible communication link between the debuggee and the frontend is used. Noosa, for example, usually runs the debuggee as a subprocess of the frontend but this is not required for communication to work. If a source-level debugger is run as the child of Noosa and that debugger is used to run the debuggee, all monitoring capabilities work as expected modulo the fact that Noosa now controls the debugger process and the debugger controls the debuggee. In practice this does not have a big impact; abstract and source-code level debugging can be mixed painlessly.

6 Related Work

The GraphTrace system [9] and tracers as described in [2] both use low-level traces of message passing activity to construct visualizations of object-oriented systems. They have been used to animate both structural and behavioural displays of programs. Both systems necessarily reveal information about object implementations to a user of the visualizations. As we have argued above this is useful in some settings but a burden in others.

The system described in [7] allows visualizations of a more conceptual nature to be built. It relies on meta-level language mechanisms such as demons to allow monitors to be separated from the code they are monitoring. In contrast to our approach, this implies that the internals of the debuggee such as data item names are visible to the visualization designer. We encapsulate this information behind debugging interfaces thereby increasing the reusability of monitors and preserving the ability to evolve the debuggee independently.

Event-based mechanisms have been widely used in algorithm animation [3,4,22]. Events in animation systems are typically hidden from the user of the system. Execution control typically consists of speed variance and an asynchronous "stop/continue" facility. Hidden events mean that it is not possible to control execution at an abstract level as we do using generalized breakpoints. Also, because algorithm animation systems are designed for visualization of

particular executions it is not possible to change program data during execution, a common debugging operation.

Some programming environments make use of events (broadcast messages) to communicate between tools; Field [17] is the most prominent example. However, Field does not use events to handle debuggee-environment interaction. The debuggee is handled by a conventional debugger which allows the placement of source-level breakpoints.

An event-based approach to monitoring is used in Dynascope [20]. Programs are executed as a combination of interpreted and compiled code. For the former, machine-level events are generated for each interpreted instruction. Abstract monitoring cannot be performed without difficulty because it requires mapping patterns of machine events into higher-level concepts.

Event-based behavioral abstraction [1] allows users to model the execution of distributed programs using events. The system lefts them compare their model with actual execution and explore differences. In contrast to our approach it does not seem possible to have objects in the program (as opposed to parts of the model) perform filtering on events. Thus any abstractions used by the program are revealed in the event stream thereby compromising encapsulation.

7 Conclusion

Profiling is a form of execution monitoring that allows programmers to examine the performance of their programs; it can help to improve that performance or even bring to light algorithm bugs. The framework described in this paper naturally extends itself to profiling. By generating events on entry to and exit from a particular object (or set of objects) it is possible to generate time-based profiles based on domain concepts rather than lower-level concepts such as functions. Useful information can also be gleaned from frequency-based profiles of event generation. We plan to add a domain-independent profiling monitor to Noosa in the near future.

In order to improve debugging for object-oriented programs we have proposed that much of the slack be taken up by the objects that form those programs rather than by the debugging tools. As well as designing objects to have some desired functionality, programmers should also take debugging into account. We have shown how it is possible to extend object interfaces to include events and extra methods to provide debugging support. Object libraries containing such support can be debugged within a powerful general framework incorporating abstract breakpoints. Program-independent execution monitors can be constructed to provide specialized views of object state and behaviour. Monitors interact with objects through abstract interfaces enhancing their reusability. Our framework can be used in conjunction with conventional debugging tools.

References

1. Bates, P. and Wileden, J. C. An Approach to High-Level Debugging of Distributed Systems. *ACM SIGPLAN Notices 18*, 8 (Aug. 1983), 107–111.
2. Böcker, H-D. and Herczeg, J. What tracers are made of. *Proc. Conf. on Object-Oriented Programming: Systems, Languages and Applications, SIGPLAN Notices 25*, 10 (Oct. 1990), 89–99.
3. Brown, M. H. *Algorithm Animation*. The MIT Press, 1988.
4. Brown, M. H. Zeus: A System for Algorithm Animation and Multi-view Editing. Digital Equipment Corporation Systems Research Center, Research Report No.75, Feb. 1992.
5. Goldberg, A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.
6. Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M. and Waite, W. M. Eli: a complete, flexible compiler construction system. *Communications of the ACM 35*, 2 (Feb. 1992), 121–131.
7. Haarslev, V. and Möller, R. A framework for visualizing object-oriented systems. *Proc. Conf. on Object-Oriented Programming: Systems, Languages and Applications, SIGPLAN Notices 25*, 10 (Oct. 1990), 237–244.
8. Ho, W. W. and Olsson, R. A. An Approach to Dynamic Linking. *Software—Practice and Experience 21*, 4 (Apr. 1991), 375–390.
9. Kleyn, M. F. and Gingrich, P. C. GraphTrace—Understanding Object-Oriented Systems Using Concurrently Animated Views. *Proc. Conf. on Object-Oriented Programming: Systems, Languages and Applications, SIGPLAN Notices 23*, 11 (Nov. 1988), 191–205.
10. Linton, M. A. The evolution of Dbx. Presented at *USENIX Summer Conference* (June 1990).

11. MONARCHI, D. E. AND PUHR, G. I. A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM 35*, 9 (Sept. 1992), 35–47.

12. O'BRIEN, P. D., HALBERT, D. C. AND KILLIAN, M. F. The Trellis Programming Environment. *Proc. Conf. on Object-Oriented Programming: Systems, Languages and Applications, SIGPLAN Notices 22*, 12 (Dec. 1987), 91–102.

13. OUSTERHOUT, J. K. Tcl: an Embeddable Command Language. Presented at *Winter USENIX Conference* (1990).

14. OUSTERHOUT, J. K. An X11 Toolkit Based on the Tcl Language. Presented at *Winter USENIX Conference* (1991).

15. PERRY, D. E. AND KAISER, G. E. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming 2*, 5 (Jan./Feb. 1990), 13–25.

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

17. REISS, S. P. Connecting tools using message passing in the Field environment. *IEEE Software 7*, 4 (July 1990), 57–66.

18. SLOANE, A. M. The structure of the GNU debugger. Department of Computer Science, University of Colorado, Boulder, CU-CS-558-1991, Nov. 1991.

19. SMITH, M. D. AND ROBSON, D. J. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming 5*, 3 (June 1992), 45–53.

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

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

22. STASKO, J. T. Tango: A Framework and System for Algorithm Animation. *Computer 23*, 9 (Dec. 1990), 27–39.

23. WAHBE, R. Efficient Data Breakpoints. *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices 27*, 9 (Sept. 1992), 200–212.

24. WIRFS-BROCK, R. J. AND JOHNSON, R. E. Surveying current research in object-oriented design. *Communications of the ACM 33*, 9 (Sept. 1990), 104–124.