Noosa: Execution Monitoring using Tcl and Tk

Anthony M. Sloane

Department of Computer Science
Campus Box 430, University of Colorado
Boulder, CO 80303-0430, USA
tony@cs.colorado.edu

Abstract

Execution monitoring is the observation of a program while it is running. Debugging and profiling are two commonly applied forms of execution monitoring. This paper describes experience using Tcl and Tk in the development of Noosa, an event-based execution monitoring system. We present an overview of the system concentrating on aspects that involve Tcl and Tk. Of particular interest is the flexibility achieved by using Tcl as the basis of both the event language and the communication between the monitoring subject and the monitors themselves.

1 Introduction

Execution monitoring [1] 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

Noosa [2] is an execution monitoring environment designed for software constructed from *reusable components*. Noosa unifies ideas from conventional debugging systems [3,4] and algorithm animation [5].

We concentrate on software constructed using reusable components such as abstract data structures, instances of abstract data types or instances of classes. This kind of software utilizes *functional interfaces* to insulate components from each other. A component's functional interface allows other components abstract access to its algorithms and data structures. Implementation details such as data representations or specifics of algorithms are hidden behind the functional interface. Components are thus able to evolve more independently than they could if such details were visible.

The central idea in Noosa is to isolate execution monitors from the details of the implementation of the subject of the monitoring. This separation simplifies monitors and insulates them from most changes in program components. We define *monitoring interfaces* that are analogous to functional interfaces except that they hide component implementations from monitors rather than from each other. In an implementation the two kinds of interfaces may be implemented using the same or different mechanisms. Noosa implements them differently because the subject and the monitors reside in different operating system processes.

Tcl [6] plays a central role in the implementation of Noosa over and above the fact that the monitors are built using Tk [7]. Section 2 describes the architecture of Noosa and explains how Tcl is used to provide powerful modes of interaction between the monitors and the subject. Section 3 illustrates this discussion with an example of a parsing monitor constructed for the Eli compiler construction system [8]. This is an example of a *domain-specific monitor*: a monitor that provides facilities for programs operating within a particular problem domain. In Section 4 we give an example of a *generic monitor* that implements breakpointing. Generic monitors are independent of the domain in which the subject operates.

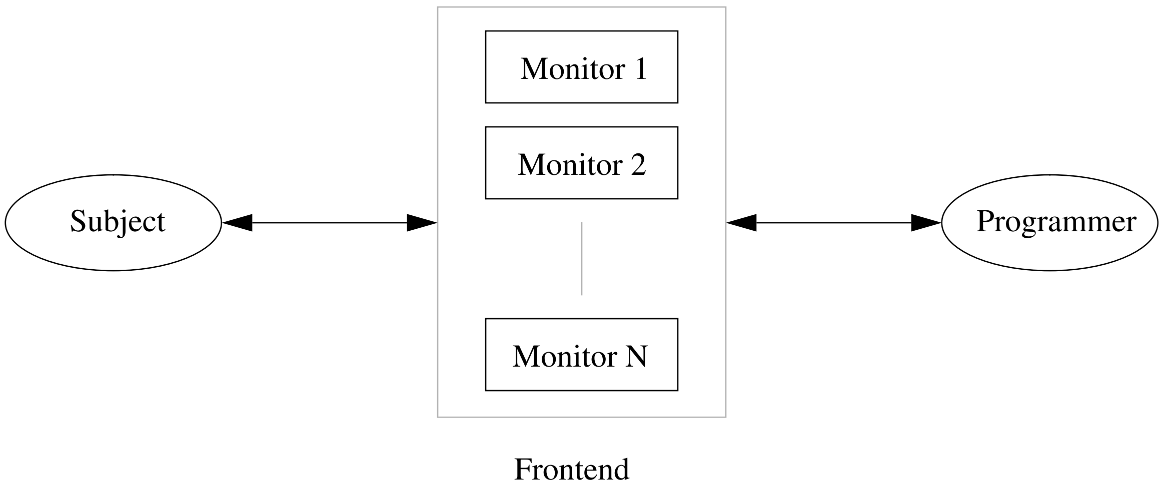

Figure 1. Noosa Architecture

The paper concludes with a brief consideration of the efficiency and usability of the system.

2 Noosa

Figure 1 shows the top-level architecture of Noosa. The programmer interacts with the *frontend* to select appropriate monitors and interacts with each monitor to specify desired monitoring operations. The monitors in turn interact with the subject during execution to implement those operations.

The main window of the Noosa frontend is shown in Figure 2. Immediately below the menu bar the current status of the subject is displayed. The “Program” entry sets the current program name and any command-line arguments that are to be used when it runs. The “Database” entry sets the Noosa database for the program (see Section 2.4). The “Monitors” pull-down menu allows the user to create instances of available monitors.

Three buttons give the user control over the execution of the subject. “Run” starts a new subject. “Continue” allows continuation from a stoppage (see Section 4). The subject can be terminated with the “Kill” button.

2.1 Communication

All communication between the subject and the frontend is performed at the level of Tcl code. Both the subject and the frontend contain Tcl interpreters. A primitive called `nsend` is used to transmit an arbitrary piece of Tcl code in either direction and return the result of evaluating it. The functionality of `nsend` is the same as the Tk primitive `send` but a named pipe implementation is used instead of communication via the X server.

2.2 The Subject

When the subject is initially run it executes normally except for event generation. Events are used to convey state change information to monitors and may be generated at arbitrary points during execution called *event generation sites*. The possible events and their semantics are not constrained by Noosa; they are chosen to match the problem domain.

Although events are theoretically enough to convey the complete state of the subject to the monitors, such an approach is impractical. In many cases it would be necessary for the monitors to duplicate the entire state of the subject just in case the user might be interested in some of it. In practice the user is only interested in a small portion of available information, so much work can be wasted.

Noosa uses *data operations* to augment events. These are arbitrary routines that can be called by

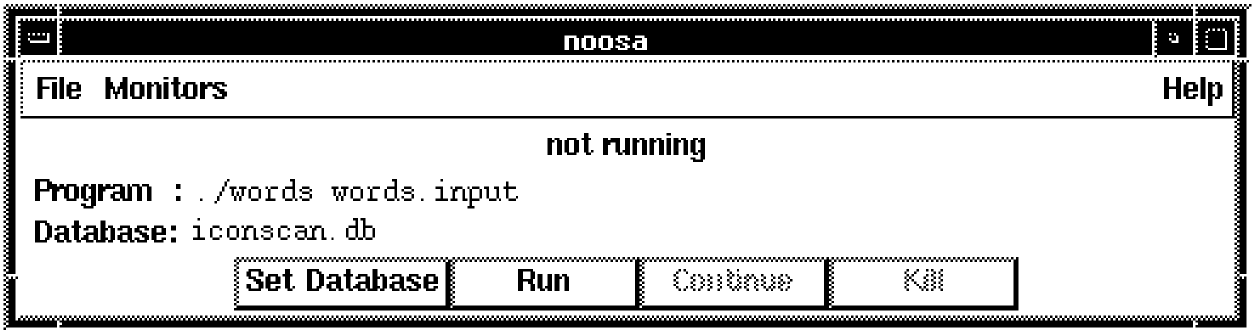

Figure 2. Subject Control Window

monitors and are executed by the subject. Operations can access or update program data.

The set of events and data operations supported by a component form its *monitoring interface*. The monitoring interface of a program is the union of the monitoring interfaces of the components from which it is constructed.

Two changes to a program are necessary to turn it into a Noosa subject:

1. Event generation sites must be identified and function calls inserted at those points to generate appropriate events. For an event type `t`, a function `generate_t` is provided. Its arguments are the attributes of the event type (if any) and are used to distinguish between instances of a single event type.
2. The program must be linked with extra code containing: a Tcl interpreter, implementations of the `generate_*` functions, and implementations of the data operations.

Event generation sites must be identified by hand. Since the target software for Noosa is based on reusable component libraries, the cost of site identification within a component can be amortized over many uses of the component. In other settings tools such as compilers can automatically insert event generation.

2.3 Monitors and Monitoring Interfaces

Monitors interact with the subject solely through monitoring interfaces. *Event handlers* can be installed in the subject by monitors to enable reactions to events. Each handler is an arbitrary piece of Tcl code that is associated with an event type and is executed whenever events of that type are generated. The values of event attributes are available to each handler as global Tcl variables. If necessary, handlers can send messages to monitors using `nsend` enabling displays to be updated and so on.

To enable monitors to control the execution of the subject, the return value of a handler is used to determine whether or not execution should continue after the current event generation. If any of the handlers for an event return one, execution is stopped at the event generation site. If all handlers return zero, execution continues. When execution stops, a synthetic `stopped` event is generated by the subject. This event can have handlers associated with it just like domain-specific events.

While execution is stopped, data operations can be invoked by monitors. Each operation is present as a Tcl procedure within the subject. `Nsend` is used to transmit a call of an operation to the subject. Once in the subject the call is executed and its value is returned to the calling monitor. Implementations of data operations are given as arbitrary C code (see the next section), so any program data can be accessed.

Two synthetic events `init` and `finit` are generated by Noosa when the subject starts and finishes execution, respectively. They enable monitors to perform initialization and finalization for each subject run.

2.4 Dapto

Dapto is a tool that largely automates the generation of the domain-specific code for a subject. It is used by the implementor of a reusable component who must design the monitoring interface for the component.

Dapto takes a specification of the monitoring interface of a component and generates the necessary code to implement event generation for events in the interface and interfaces to its data operations. Event types are specified by giving their signatures, that is, the name of the event type, and the names and types of its attributes. Data operations are given by their signatures and their bodies. The latter are arbitrary fragments of C code to implement the operation. Normally this C code accesses program data structures to implement the operation. Section 4 contains an example of a Dapto specification.

From a monitoring interface specification Dapto generates the following:

1. A C implementation of the `generate_t` function for each event type `t`. These functions store the values of the event attributes in global Tcl variables, call the event handlers and either return to normal program execution or suspend execution depending on the return values of the event handlers.
2. C implementations of a Tcl command procedure for each data operation. The implementation of an operation consists of the C code provided in the specification augmented with a generated test to check the validity of its argument list.
3. Initialization code to install the data operation command procedures into the subject's interpreter as primitives.
4. Tcl code representing a database of information about the monitoring interface. This code is loaded by the frontend and enables it to decide which monitors are applicable to a subject and lets monitors display interface information (see Section 4).

The C code generated by Dapto is compiled with the regular code for the program to form an executable for the subject.

3 Parsing Monitor

Eli [8] generates compilers from very high-level specifications of their functionality. Eli incorporates two LALR(1) parser generators. This section briefly describes a monitor for the components generated by these tools. Because the monitor communicates with the components via a well-defined monitoring interface it is able to work with the outputs of either of the tools.

Parsing is the process of determining the structure of an input text given a stream of tokens from that input text produced by a lexical analyzer. Eli allows text structure to be described by context-free grammars. The parsing monitor allows the relationship between a context-free grammar and a given input text to be monitored. This allows incorrect structuring to be easily diagnosed.

Figure 3 shows a typical view of the Eli parsing monitor constructed using Noosa. Selecting a location in the input text (lower text window) causes the upper text window to display the context-free productions (if any) that were used to recognize that text location. In this case the highlighted `x` identifier was selected.

Displayed productions range from most general at the top to most specific at the bottom. Thus the first production displayed 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` inside a `VariableNameUse` contained in 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 `IfStatement` shows that the following `AssignmentStatement` is in the `then`-clause rather than the `else`-clause.

Selecting a production instance in the upper window will highlight the extent in the input text that was recognized by that production instance.

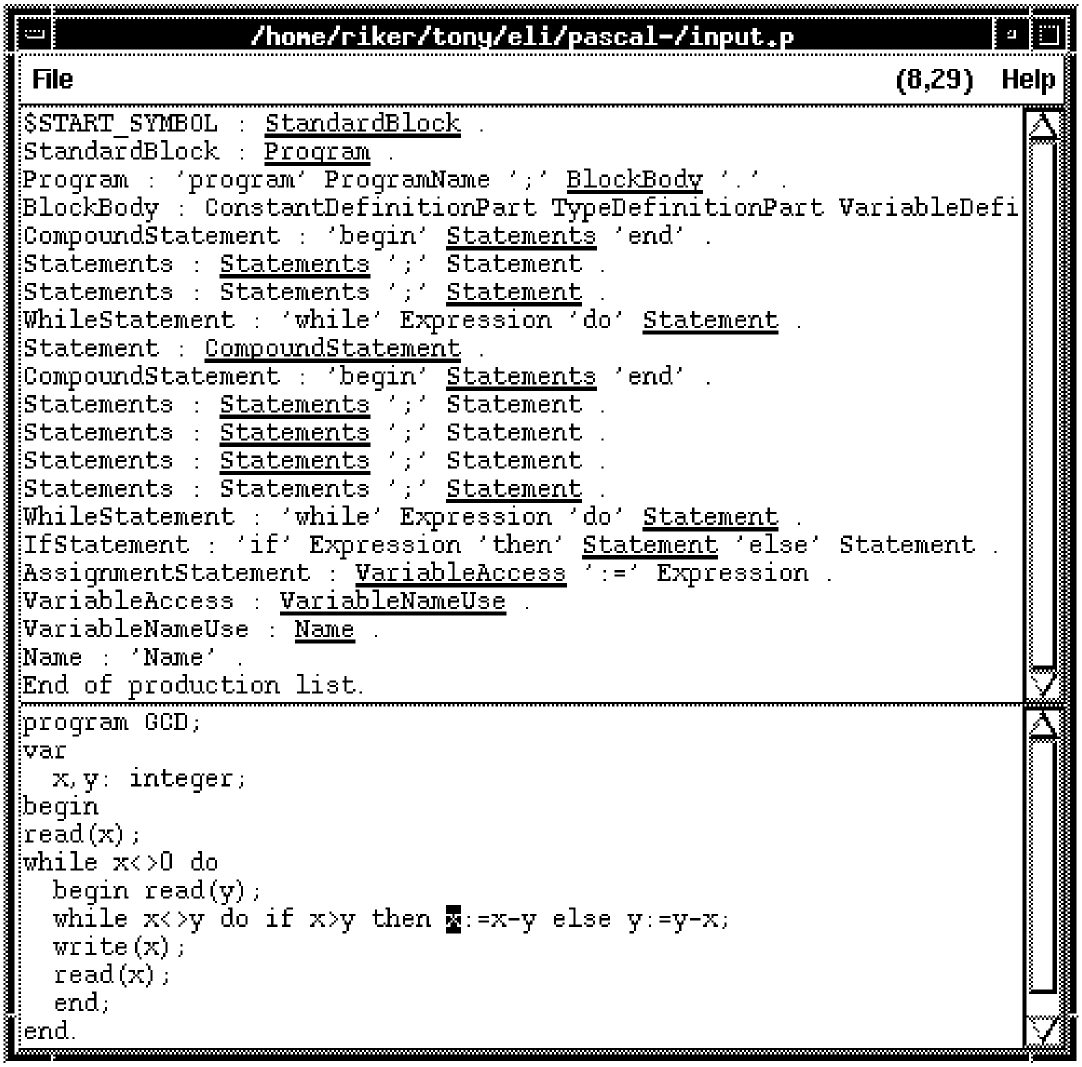

Figure 3. Eli Parsing Monitor

3.1 Monitoring Interface

The Eli parser generators have been modified to provide a simple monitoring interface that supports the parsing monitor. These changes represent less than one per cent of the code of each tool.

A `recognition` event is generated whenever a piece of input text is recognized. Also a `get_conc_prod` operation provides access to the productions in the concrete grammar used to generate the parser. The complete Dapto specification for this interface is given in Figure 4. The implementation of `get_conc_prod` uses a table of productions produced by the parser generator. In Figure 4 quoted strings are used for documentation purposes (see the next section).

The handler used by the parsing monitor to react to `recognition` events is:

```
nsend parse_recog $prod $uses \
    $linebeg $colbeg $lineend \
    $colend
```

This simply sends the attribute values to the monitor where the Tcl procedure `parse_recog` stores them for later use.

Other monitors have been built dealing with the following aspects of Eli-generated programs: string storage, lexical analysis, name analysis, message generation, scoping, symbol table maintenance, and semantic analysis (attribute grammar monitoring). Domains other than Eli have also been investigated. Noosa was used to

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

operation get_conc_prod "Retrieve text of a concrete production"
   (int index "Index of the production"): str
{
   extern char *conc_prods[];
   sprintf (interp->result, "%s",conc_prods[index]);
}
```

Figure 4. Monitoring Interface for Parsing Monitor

build a string scanning monitor for the Icon programming language [9]. Currently a monitor for memory leaks in C programs is being constructed. Details of these monitors will appear in [2].

4 Breakpoints

Execution control in conventional debugging systems is performed via *breakpoints* [3,4]. Each breakpoint is associated with a source code location and possibly other information such as context conditions or counts. A breakpoint triggers when its location is reached during execution and (say) the condition is true or decrementing the count yields zero. Data breakpoints are variants that allow conditions to be implicitly tested at all locations in the program at once. Usually when a breakpoint of any kind triggers, execution is stopped or a sequence of user-defined debugger commands are executed.

The Noosa system can be used to provide sophisticated breakpointing capabilities. Because Noosa is designed to hide the source code of program components from monitors it is not possible to attach breakpoints to source locations. Instead breakpoints are attached to event types giving them an abstract feel.

Breakpoints can be achieved by specifying event handlers that return one. Recall that execution of the subject is suspended if any of the handlers for an event return one. Thus the following handler for `recognition` events will cause execution to stop as soon as a piece of text past line five is recognized:

```
return ($linebeg > 5)
```

Because handlers execute in a full Tcl interpreter in the subject they can use any Tcl facility. For example, they can use global variables to communicate. For example, the handler:

```
global last
set ret ($prod == 12 && \
         $last == 1)
set last $prod
return $ret
```

will cause a stoppage whenever production twelve is recognized immediately after production one. Similar techniques can be used to implement temporary breakpoints that can only be triggered once, or counting breakpoints that are triggered after a set number of times. Data breakpoints are trivial providing the monitoring interface includes an event that represents changes to the data of concern.

To simplify the setting of breakpoints, Noosa provides a generic breakpoint monitor (Figure 5). A generic monitor is one that does not depend on the semantics of events or data operations. The breakpoint monitor lets users specify arbitrary handlers for events. The monitor itself has no knowledge of the meaning of any of the events. This is in contrast to domain-

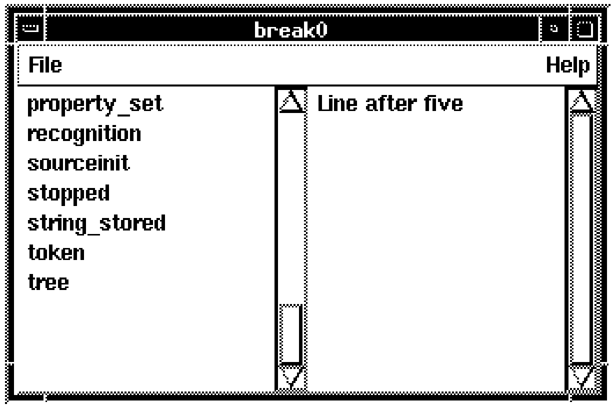

Figure 5. Noosa Breakpoint Monitor

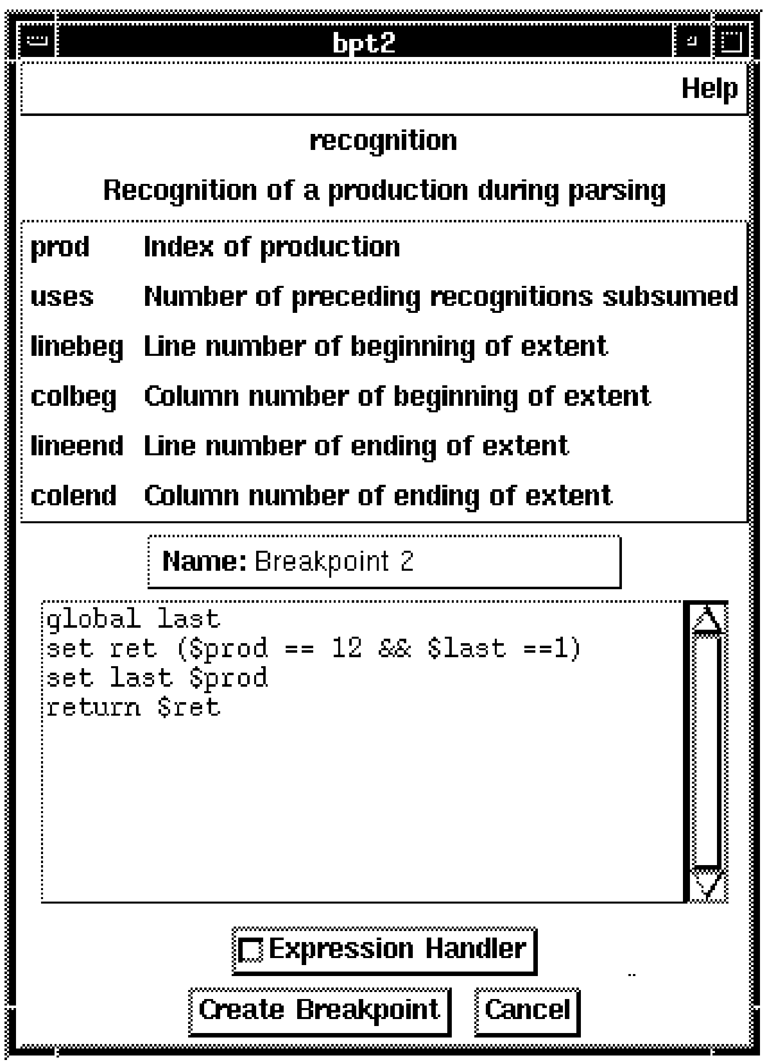

Figure 6. Setting a Breakpoint

specific monitors like the parsing monitor described in Section 3 which placed a semantic interpretation on `recognition` events.

Figure 5 shows the breakpoint monitor after the first example breakpoint above has been set. The user has to named the breakpoint “Line after five”. Breakpoints can also be disabled or deleted using this interface.

Figure 6 shows the user in the process of setting the second breakpoint. The documentation strings from the Dapto specification are displayed as aids. The text window is used to insert the handler code. A similar interface allows breakpoints to be altered.

The default mode of the monitor (not shown) is to have the “Expression Handler” checkbutton on. In this mode the user can just enter a Tcl expression for the breakpoint condition more closely approximating traditional breakpoint facilities. The monitor will wrap the following code around the expression to form the handler:

```
if {expression} {
    return 1;
}
```

Other generic monitors have been constructed. Time and frequency profiles were built using event timestamps and counts. Similarly, event transcripts are easily obtained and are useful in some monitoring situations.

5 Conclusion

Tcl and Tk have enabled an extremely flexible monitoring environment to be constructed in a short period of time. Using Tk to build the graphical interfaces to monitors permitted presentation ideas to be prototyped quickly. Having Tcl as the basis of the event and data operation mechanisms has resulted in a simple but powerful facility. Although detailed performance analysis is yet to be conducted, Noosa operates fast enough to enable monitoring to be done effectively. There is room for improvement in the event generation process.

The Eli monitors have greatly improved the development process for Eli programs. Previously users had to rely on using debuggers such as GDB or Dbx on the Eli-generated code. This required extensive knowledge of the internals of this code which was either unavailable to most Eli users or time-consuming to obtain and soon out-of-date. Noosa allows developers of tools used in Eli to build monitoring interfaces that isolate monitors (and hence users) from the details of their tools. It is now possible to monitor Eli-generated programs at the level of user specifications rather than generated code. Application to other problem domains is showing that the techniques have general applicability and utility.

References

[1] B. Plattner, J. Nievergelt. Monitoring program execution: a survey. *Computer*, 14(11), pages 76-93, November 1981.

[2] A. M. Sloane. Domain-level execution monitoring. Ph.D. Thesis, University of Colorado, Boulder. 1993. In preparation.

[3] R. M. Stallman, R. H. Pesch. *The GNU source-level debugger.* Free Software Foundation. 1993.

[4] M. A. Linton. The evolution of Dbx. *USENIX Summer Conference*, pages 211-220, 1990.

[5] M. H. Brown. *Algorithm animation*. The MIT Press, 1986.

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

[7] J. K. Ousterhout. An X11 toolkit based on the Tcl language. *USENIX Winter Conference.* 1991.

[8] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, W. M. Waite. Eli: a complete, flexible compiler construction system. Communications of the ACM, 35(2), pages 121-131, February 1992.

[9] R. E. Griswold, M. T. Griswold. *The Icon Programming Language.* Prentice-Hall. 1983.