The Design and Implementation of a Dataflow Language for Scriptable Debugging[*][†]

Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro,
Shriram Krishnamurthi, Steven P. Reiss
Computer Science Department
Brown University
115 Waterman Street, Providence, RI 02912, USA

January 17, 2006

Abstract

Debugging is a laborious, manual activity that often involves the repetition of common operations. Ideally, users should be able to describe these repetitious operations as little programs. Debuggers should therefore be programmable, or *scriptable*. The operating environment of these scripts, however, imposes interesting design challenges on the programming language in which these scripts are written.

This paper presents our design of a language for scripting debuggers. The language offers powerful primitives that can precisely and concisely capture many important debugging and comprehension metaphors. The paper also describes a pair of debuggers, one for Java and the other for Scheme, built in accordance with these principles. The paper includes concrete examples of applying this debugger to programs.

1 Introduction

Debugging is a laborious part of the software development process. Its unpleasantness is exacerbated by many contemporary debuggers, which offer only primitive capabilities. Indeed, even with the growing sophistication of visual programming environments, the underlying debugging tools remain fairly primitive.

Debugging is a complex activity because there is often a good deal of knowledge about a program that is not explicitly represented in its execution. For instance, imagine a programmer trying to debug a large data structure that appears not to satisfy an invariant. He might set a breakpoint, examine a value, compare it against some others and,

[*]This research was partially supported by NSF grant CCR-0305949.
[†]A preliminary version of this paper appeared in the 19th IEEE International Conference on Automated Software Engineering (ASE 2004) [24].

not finding a problem, resume execution, perhaps repeating this process dozens of times. This is both time-consuming and dull; furthermore, a momentary lapse of concentration may cause him to miss the bug entirely.

The heart of automated software engineering lies in identifying such repetitive human activities during software construction and applying computational power to ameliorate them. For debuggers, one effective way of eliminating repetition is to make them *scriptable*, so users can capture common patterns and reuse them in the future. The problem then becomes one of designing effective languages for scripting debuggers.

Debugging scripts must easily capture the programmer's intent and simplify the burdensome aspects of the activity. To do this, they must meet several criteria. First, they must match the temporal, event-oriented view that programmers have of the debugging process. Second, they must be powerful enough to interact with and monitor a program's execution. Third, they should be written in a language that is sufficiently expressive that the act of scripting does not become onerous. Finally, the scripting language must be practical: users should, for instance, be able to construct *program-specific* methods of analyzing and comprehending data. For example, users should be able to create redundant models of the program's desired execution that can be compared with the actual execution. This calls for a library of I/O and other primitives more commonly found in general-purpose languages than in typical domain-specific languages.

In this paper, we present the design and implementation of an interactive scriptable debugger called MzTake (pronounced "miz-take"). Predictably, our debugger can pause and resume execution, and query the values of variables. More interestingly, developers can write scripts that automate debugging tasks, even in the midst of an interactive session. These scripts are written in a highly expressive language with a dataflow evaluation semantics, which is a natural fit for processing the events that occur during the execution of a program. In addition, the language has access to a large collection of practical libraries, and evaluates in an interactive programming environment, DrScheme.

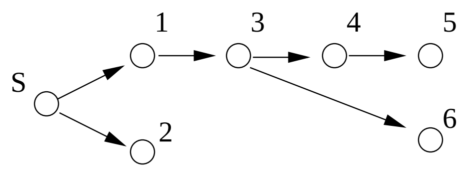
2 A Motivating Example

Figure 1 shows a Java transcription of Dijkstra's algorithm, as presented in *Introduction to Algorithms* [8]. Recall that Dijkstra's algorithm computes the shortest path from a source node to all the other nodes in a graph. It is similar to breadth-first search, except that it enqueues the nodes according to the total *distance* necessary to reach them, rather than by the number of *steps*. The length of the shortest path to a node (so far) is stored in the *weight* field, which is initialized to the floating point infinity. The algorithm relies on the fact that the shortest-path estimate for the node with the smallest weight is provably optimal. Accordingly, the algorithm removes that node from the pool (via *extractMin*), then uses this optimal path to improve the shortest path estimate of adjacent nodes (via *relax*). The algorithm makes use of a priority queue, which we also implemented.

Figure 2 shows a concrete input graph (where S, at location $\langle 100, 125\rangle$, denotes the source from which we want to compute distances) and the output that results from executing this algorithm on that graph. The output is a set of nodes for which the algorithm was able to compute a shortest path. For each node, the output presents the node's number, its coordinates, and its distance from the source along the shortest path.

```
class DijkstraSolver {

  public HashMap backtrace = new HashMap();
  private PriorityQueue q = new PriorityQueue();

  public DijkstraSolver(DirectedGraph graph,
                        Node source) {
    source.weight = 0.0;
    q.addAll(graph.getNodes());

    while(!q.isEmpty()) {
      Node node = (Node)q.extractMin();
      List successors = graph.getSuccsOf(node);
      for(Iterator succIt = successors.iterator();
          succIt.hasNext(); )
        relax(node, (Node)succIt.next());
    }
    System.out.println("Result␣backtrace:\n" +
                       backtrace.keySet());
  }

  public void relax(Node origin, Node dest) {
    double candidateWeight =
      origin.weight + origin.distanceTo(dest);
    if (candidateWeight < dest.weight) {
      dest.weight = candidateWeight;
      backtrace.put(dest, origin);
    }
  }
}
```

Figure 1: Implementation of Dijkstra’s Algorithm



```
Result backtrace:
[[node 1 : x 150 y 100 weight 55],
 [node 2 : x 150 y 150 weight 55],
 [node 3 : x 200 y 100 weight 105]]
```

Figure 2: Sample Input and Output

As we can see, this output is incorrect. The algorithm fails to provide outputs for the nodes numbered 4, 5 and 6, even though the graph is clearly connected, so these are a finite distance from S.

Since the implementation of Dijkstra's algorithm is a direct transcription from the text (as a visual comparison confirms), but *we* implemented the priority queue, we might initially focus our attention on the latter. Since checking the overall correctness of the priority queue might be costly and difficult, we might first try to verify a partial correctness criterion. Specifically, if we call *extractMin* to remove two elements in succession, with no insertions in-between, the second element should be at least as large as the first.

Unfortunately, most existing debuggers make it difficult to automate the checking of such properties, by requiring careful coordination between breakpoint handlers. For example, in `gdb` [28] we can attach conditional breakpoint handlers—which are effectively callbacks—to breakpoints on *insert* and *extractMin*, and so observe values as they enter and leave the queue. Figure 3 illustrates the control flow relationship between the target and the debugging script when we use callbacks to handle events. Starting at the top left, the target program runs for a while until it reaches the *extractMin* function; control then shifts to the debugger, which invokes the callback. The callback makes a decision to either pause or resume the target. Eventually, the target continues and runs until it reaches the breakpoint on the *extractMin* function for a second time. If we are monitoring a temporal property, such as the ordering of elements taken out of a priority queue, the decision to pause or resume the target on the second interruption will depend on data from the first callback invocation. Observe that, for the program on the left, it is natural to communicate data between the parts of execution, because it consists of one single thread of control. In contrast, the "program" on the right is broken up into many disjoint callback invocations, so we need to use mutable shared variables or other external channels to communicate data from one invocation to the next.

All this is simply to check for pairs of values. Ideally, we want to go much further than simply checking pairs. In fact, we often want to create a redundant model of the execution, such as mirroring the queue's intended behavior, and write predicates that check the program against this model. Upon discovering a discrepancy, we might want to interactively explore the cause of failure. Moreover, we might find it valuable to abstract over these models and

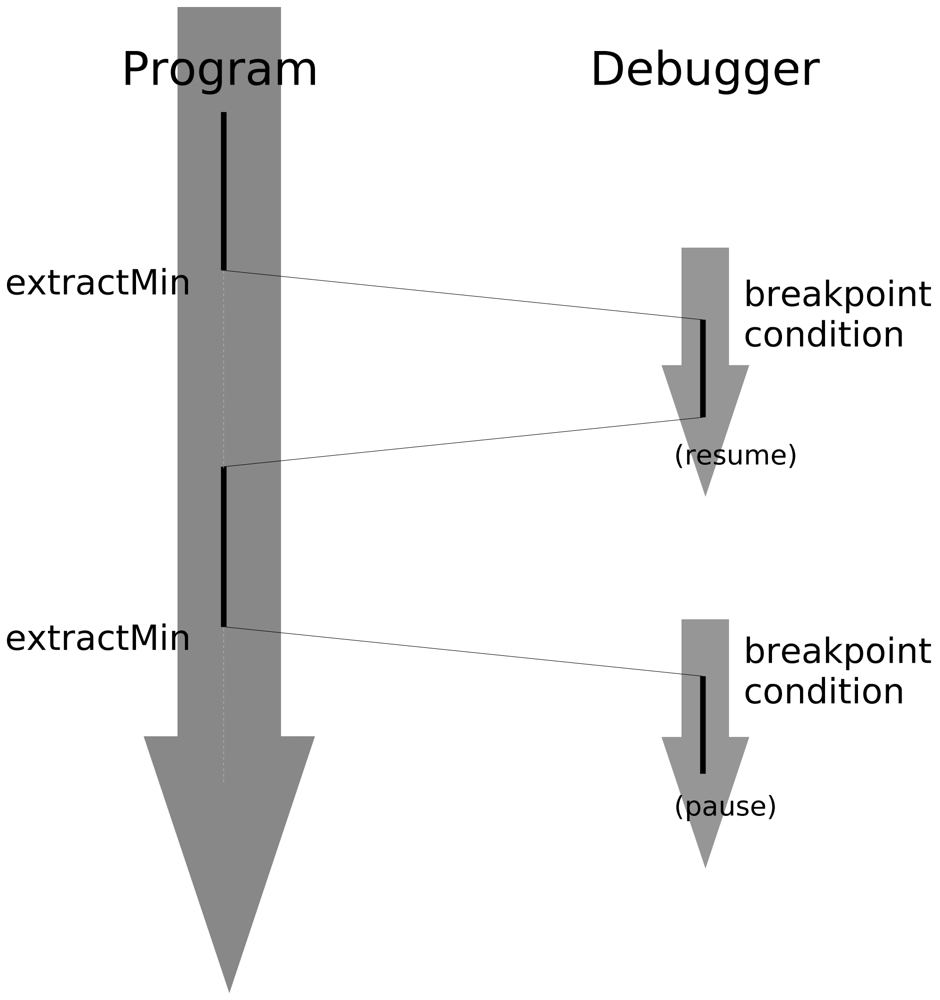


Figure 3: Control Flow of Program and Script

predicates, both to debug similar errors later and to build more sophisticated models and predicates as the program grows in complexity.

In principle, this is what scriptable debugging should accomplish well. Unfortunately, this appears to be difficult for existing scriptable debuggers. For example, Coca [12] offers a rich predicate language for identifying interesting data and points in the execution, but it does not offer a facility for relating values across different points in time, so the programmer would still need to monitor this criterion manually. UFO [4] supports computation over event-streams, but does not support interaction. Dalek [26] is interactive and offers the ability to relate execution across time, but provides limited abstractions capabilities, so we could not use it to build the predicates described in this paper. In general, existing scriptable debuggers appear to be insufficient for our needs; we discuss them in more detail in section 11.

This paper presents our new language and infrastructure that address the weaknesses found in existing debuggers. In section 3, we describe the goals and observations that have guided our work, and in section 4, we introduce the dataflow language FrTime (pronounced "father time"), on top of which we have built MzTake. We reflect on lessons learned from this example in section 6. In Section 7 and Section 8, we describes the design and the implementation, respectively. Section 9 discusses strategies to control the execution of a target program. Section 10 provides additional, illustrative examples of the debugger's use.

3 Desiderata

We believe that users fundamentally view debugging as a temporal activity with the running program generating a stream of events (entering and exiting methods, setting values, and so on). They use constructs such as breakpoints to make these events manifest and to gain control of execution, at which point they can inspect and set values before again relinquishing control to the target program. To be maximally useful and minimally intrusive, a scriptable debugger should view the debugging process just as users do, but make it easy to automate tedious activities.

Concretely, the scripting language must satisfy several important design goals.

1. While debuggers offer some set of built-in commands, *users often need to define problem-specific commands*. In the preceding example, we wanted to check the order of elements extracted from a queue; for other programs, we can imagine commands such as "verify that this tree is balanced". While obviously a debugger should not offer commands customized to specific programs, it should provide a powerful enough language for programmers to capture these operations easily. Doing so often requires a rich set of primitives that can model sophisticated data, for instance to track the invariants of a program's data.

2. Programs often contain implicit invariants. Validating these invariants requires maintaining auxiliary data structures strictly for the purpose of monitoring and debugging. In our example, although Dijkstra's algorithm depends on nodes being visited in order of weight, there is no data structure in the program that completely

captures the ordered list of nodes (a priority heap satisfies only a weaker ordering relation). Lacking a good debugging framework, the developer who wants to monitor monotonicity therefore needs to introduce explicit data structures into the source. These data structures may change the space- and time-complexity of the program, so they must be disabled during normal execution. All these demands complicate maintenance and program comprehension. Ideally, *a debugger should support the representation of such invariants outside the program's source*. (In related work, we explain why approaches like contracts and aspects [3] are insufficient.)

3. Debugging is often a process of generating and falsifying hypotheses. *Programmers must therefore have a convenient way to generate new hypotheses while running a program.* Any technique that throws away the entire debugging context between each attempt is disruptive to this exploratory process.

4. Since the target program is a source of events and debugging is an event-oriented activity, *the scripting language must be designed to act as a recipient of events*. In contrast, traditional programming languages are designed for writing programs that are "in control"—i.e., they determine the primary flow of execution, and they provide cumbersome frameworks for processing events. This poses a challenge for programming language design.

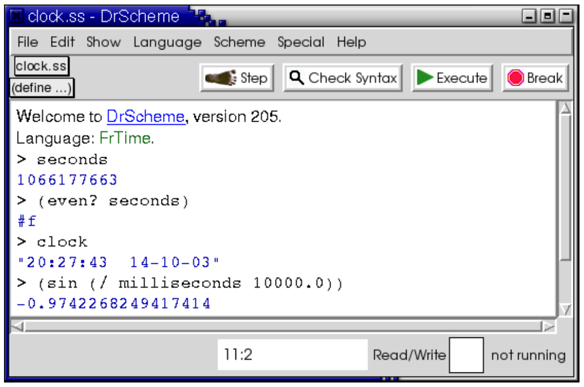
5. As a pragmatic matter, *debuggers should have convenient access to the rich* I/O *facilities provided by modern consoles* so they can, for instance, implement problem-specific interfaces. A custom language that focused solely on the debugging domain would invariably provide only limited support for such activities. In contrast, the existence of rich programming libraries is important for the widespread adoption of a debugging language.

To accomplish these goals, a debugging language must address a conflict central to all language design: balancing the provision of powerful abstractions with restrictions that enable efficient processing. This has been a dominant theme in the prior work (see section 11). Most prior solutions have tended toward the latter, while this paper begins with a general-purpose language, so as to explore the space of expression more thoroughly. This results in some loss of machine-level efficiency, but may greatly compensate for it by saving users' time. Furthermore, the functional style we adopt creates opportunities for many traditional compiler optimizations.

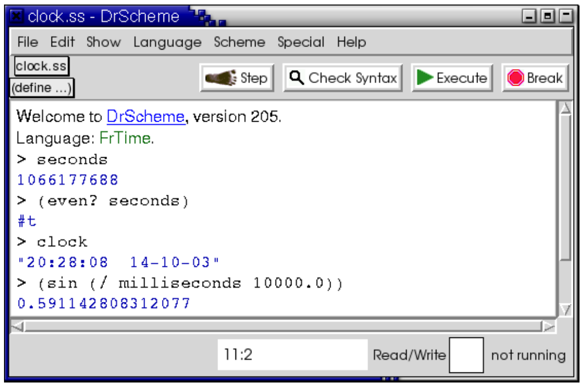
4 The FrTime Programming Language

Instead of implementing our debugging language from scratch, we have built it atop the language FrTime—a dataflow language (with Lisp-based syntax) that supports stateful operations and provides a library of data structures and primitives for most common programming activities [7]. The language is inspired by work on Functional Reactive Programming [13], whose motivation is to allow declarative expression of reactive systems.

The motivation for FrTime is easy to explain with a simple example. Most programming languages have a library primitive for accessing the current time. A variable that holds the response from this primitive is, however, assigned only when the primitive returns; the value becomes outdated as execution continues, unless the program explicitly



```
Welcome to DrScheme, version 205.
Language: FrTime.
> seconds
1066177663
> (even? seconds)
#f
> clock
"20:27:43  14-10-03"
> (sin (/ milliseconds 10000.0))
-0.9742268249417414
```



```
Welcome to DrScheme, version 205.
Language: FrTime.
> seconds
1066177688
> (even? seconds)
#t
> clock
"20:28:08  14-10-03"
> (sin (/ milliseconds 10000.0))
0.591142808312077
```

Figure 4: Screenshots of FrTime in Action

performs operations to keep the value current. In contrast, FrTime provides built-in support for *time-varying values*, called *behaviors*, that automatically update with the passage of time. For instance, the expression *seconds* is a built-in behavior whose value updates every second in lockstep with the system clock.

Any expression that uses a time-varying value itself becomes time-varying. For instance, the expression (*even? seconds*) changes every time the value of *seconds* changes (i.e., every second), alternating between the values true and false. The implementation of FrTime is responsible for automatically tracking dependencies between primitive signals and expressions that depend on them, ordering these dependencies, forcing fresh computation, and propagating values whenever they change. Behaviors can take value *undefined*, which acts as a bottom. Any operation applied to *undefined* also returns *undefined*.

FrTime offers run-time support through the DrScheme programming environment [15]. Firstly, the rich libraries of DrScheme are available for FrTime, and are automatically lifted to the time domain, so they recompute when their arguments update. Secondly, the DrScheme prompt recognizes behaviors and automatically updates the display of their values as they change over time. Figure 4 shows the same DrScheme session, displaying several FrTime expressions, captured (untouched) twenty-five seconds apart. The expressions we evaluated have not changed, but their answers have—the last three digits of the Unix time progressed from “663” to “688”, and the displayed values updated accordingly. Indeed, values returned from FrTime expressions are animated, correctly representing their time-varying nature.

In addition to behaviors, FrTime also has *events*. Whereas behaviors have a value at any point in time, events are discrete: for instance, the event *key-strokes* is an infinite stream that yields a new value every time a key is pressed. FrTime provides a set of functional combinators that process event-streams; for instance, *hold* converts event-streams into behaviors by consuming an event-stream and returning a behavior whose value is always the most recent value in the stream. Thus, (*hold key-strokes* 'none-yet) is a behavior whose value is initially the symbol 'none-yet and, from the first keystroke onwards, the value of the last key pressed.

FrTime upholds a number of guarantees about a program's execution, including the order in which it processes

```
(define c (start-vm "DijkstraTest"))
(define queue (jclass c PriorityQueue))

(define inserts
  (trace ((queue . jdot . add) . jloc . entry)
        (bind (item) (item . jdot . weight))))
(define removes
  (trace ((queue . jdot . extractMin) . jloc . exit)
        (bind (result) (result . jdot . weight))))

(define violations
  (not-in-order (merge-e removes (inserts . -=> . 'reset))))
(define latest-violation (hold violations false))
(define (nv)
  (set-running-e! (violations . -=> . false)))
```

Figure 5: Monitoring the Priority Queue

events and the space required to do so:

- **Ordering of event processing**: Since FrTime must listen to multiple concurrent event sources and recompute various signals in response, we might worry about the possibility of timing and synchronization issues. For example, if signal *a* depends on signal *b*, we would like to know that FrTime will not recompute *a* using an out-of-date value from *b*. Fortunately, FrTime's recomputation algorithm is aware of dataflow dependencies between signals and updates them in a topological order, starting from the primitive signals and working towards their dependents.

- **Space consumption**: FrTime only remembers the current values of behaviors and the most recent occurrences of events. Thus, if the program's data structures are bounded, then the program can run indefinitely without exhausting memory. If the application needs to maintain histories of particular event streams, it can use FrTime primitives like *history-e* or *accum-b* for this purpose. The application writer must apply these operations explicitly and should therefore be aware of their cost.

The interested reader can learn more about the language from a companion paper [7] or by experimenting with the implementation, which is part of the DrScheme distribution.

5 Debugging the Motivating Example

We are now ready to return to our example from section 2. As we explained previously, our implementation of Dijkstra's algorithm employs a priority queue coded by us. In addition, we noted that our implementation of *DijkstraSolver* is a direct transcription of the pseudocode in the book. We hypothesized that the bug might be in the implementation of the priority queue, and that we should therefore monitor its behavior. Recall that the partial correctness property we wanted to verify was that consecutive pairs of elements extracted from the queue are in non-decreasing order.

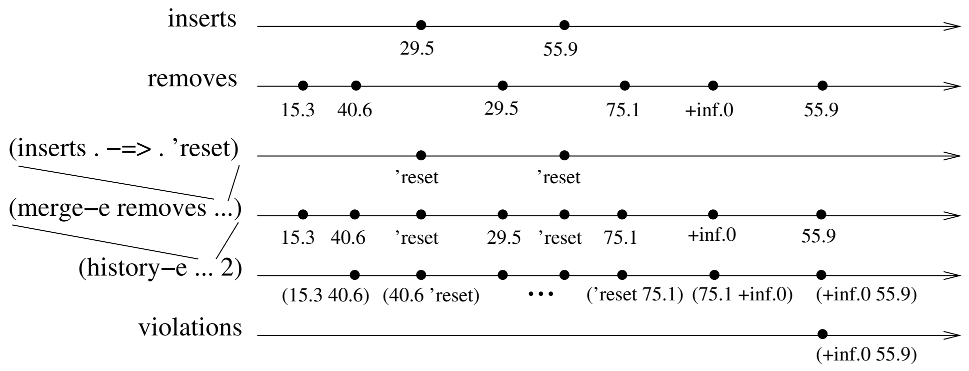


Figure 6: Event Streams

Figure 5 presents a debugging script that detects violations of this property. In the script, the variable *c* is bound to a debugging session for *DijkstraTest*, a class that exercises the implementation of Dijkstra's algorithm. The invocation of *start-vm* initiates the execution of the Java Virtual Machine (JVM) on this class, and immediately suspends its execution pending further instruction.

The expression (**jclass** *c PriorityQueue*) creates a FrTime proxy for the *PriorityQueue* class in Java. Since Java dynamically loads classes on demand, this proxy is a time varying value: its value is *undefined* at first, and stays so until the class is loaded into the JVM. The operator **jclass** treats its second argument specially: *PriorityQueue* is not a variable reference, but simply the name of the target class. In Lisp terminology, **jclass** is a *special form*. So are **jdot** (which returns the value of a field) and **jloc** (which selects a location within a method).

Next, we install tracing around the methods *add* and *extractMin* of the priority queue. A *tracepoint* is a FrTime event-stream specifically designed for debugging: the stream contains a new value every time the Java program's execution reaches the location marked by the tracepoint. Concretely, the expression

```
(define inserts
  (trace ((queue . jdot . add) . jloc . entry)
         (bind (item) (item . jdot . weight))))
```

installs a tracepoint at the entry of the *add* method of *queue*.[1] The result of **trace** is an event stream of values. There is an event on the stream each time the target program reaches the *add* method. To generate the values in the stream, the **trace** construct evaluates its body; this body is re-evaluated for each event. In this instance, we use the **bind** construct to reach into the stack of the target, find the value of the variable *item* (in the target), and bind it to the identifier *item* (in the body of the **bind**). In turn, the body of the **bind** extracts the *weight* field from this item. This weight becomes the value of the event.

The identifier *inserts* is therefore bound to a FrTime event-stream consisting of the weights of all nodes inserted

[1]Here and in the rest of this paper, we use the infix notation supported by FrTime: (*x* . *op* . *y*) is the same as (*op x y*) in traditional Lisp syntax.

```
(define (not-in-order e)
  (filter-e
   (match-lambda
     [('reset _) false]
     [(_ 'reset) false]
     [(previous current) (> previous current)])
   (history-e e 2)))
```

Figure 7: The Monitoring Primitive

```
(define inserters
  (inserts . ==> . insert-in-model))
(define removers
  (removes . ==> . remove-from-model))

(define model
  (accum-b (merge-e inserters removers)
           (convert-queue-to-list (bind (q) q))))
```

Figure 8: The Redundant Model

into the priority queue. The identifier *removes* is bound correspondingly to the weights of nodes removed from the queue by *extractMin*.

We initially want to perform a lightweight check that determines whether consecutive *remove*s (not separated by an *insert*) are non-decreasing. To do this, we merge the two event-streams, *inserts* and *removes*. Since we are only interested in consecutive, uninterrupted removals, the monitor resets upon each insertion. The following FrTime code uses the combinator -=> to map the values in the *inserts* stream to the constant 'reset, which indicates that the monitor should reset:

```
(merge-e removes (inserts . -=> . 'reset))
```

The result of this expression is illustrated in Figure 6. In this graph, time flows towards the right, so earlier events appear to the left. Each circle represents one event occurrence on the corresponding stream. The first three lines show the streams we just discussed: *inserts*, *removes*, and the mapped *inserts*. The fourth timeline of the figure shows that the *merge-e* expression evaluates to an event-stream whose events are in the order they are encountered during the run. The insert events have been mapped to the constant, while the remove events are represented by the weight of the node.

The last two timelines in Figure 6 depict the next two streams created by the script. The merged stream is passed to the core monitoring primitive, *not-in-order*, shown in Figure 7. This uses *history-e* to extract the two most recent values from the stream and processes each pair in turn. It filters out those pairs that do not exhibit erroneous behavior, namely when one of the events is a 'reset or when both events reflect extracted weights that are in the right order. The result is a stream consisting of pairs of weights where the weightier node is extracted first, violating the desired order. We call this stream *violations*.

The FrTime identifier *latest-violation* is bound to a behavior that captures the last violation (using the FrTime combinator *hold*). If the priority queue works properly, this behavior will retain its initial value, false (meaning "no violation so far"). If it ever changes, we want to pause the JVM so that we can examine the context of the violation.

To do this, we use the primitive **set-running-e!**, which consumes a stream of boolean values. Calling **set-running-e!** launches the execution of the target program proper, and it will keep on consuming future events on the given stream: when an event with the value false occurs the JVM pauses, after which, when an event with a true value occurs the JVM resumes.[2] Since we anticipate wanting to observe numerous violations, we define the (concisely named) abstraction *nv*, which tells the JVM to run until the **n**ext **v**iolation occurs.

At the interactive prompt, we type (*nv*). Soon afterward, the JVM stops, and we query the value of *latest-violation*:

```
> (nv)
    short pause
> latest-violation
(+inf.0   55.90169943749474)
```

This output indicates that the queue has yielded nodes whose weights are out of order. This confirms our suspicion that the problem somehow involves the priority queue.

Continuing Exploration Interactively

To identify the problem precisely, we need to refine our model of the priority queue. Specifically, we would like to monitor the queue's complete black-box behavior, which might provide insight into the actual error.

With the JVM paused, we enter the code in figure 8 to the running FrTime session. This code duplicates the priority queue's implementation using a sorted list. While slower, it provides redundancy by implementing the same data structure through an entirely different technique, which should help identify the true cause of the error.[3]

We now explain the code in figure 8. The identifier *model* is bound to a list that, at every instant, consists of the elements of the queue in sorted order. We decompose its definition to improve readability. The value *inserters* is an event-stream of FrTime procedures that insert the values added to the priority queue into the FrTime model (==> applies a given procedure to each value that occurs in an event-stream); similarly, *removers* is bound to a stream of procedures that remove values from the queue. The code

```
(accum-b (merge-e inserters removers)
         (convert-queue-to-list (bind (q) q)))
```

merges the two streams of procedures using *merge-e*, and uses *accum-b* to apply the procedures to the initial value of the model. *accum-b* accumulates the result as it proceeds, resulting in an updated model that reflects the application of all the procedures in order. *accum-b* returns a behavior that reflects the model after each transformation. We must

[2]In Scheme, any value other than false is true.

[3]Since the property we are monitoring depends only on the nodes' weights, not their identities, the model avoids potential ordering discrepancies between equally-weighted nodes.

initialize the model to the current content of the queue. The user-defined procedure *convert-queue-to-list* (elided here for brevity) converts *q*'s internal representation to a list.

Having installed this code and initialized the model, we resume execution with *nv*. At the next violation, we interactively apply operations to compare the queue's content against its FrTime model (the list). We find that the queue's elements are not in sorted order while those in the model are. More revealingly, the queue's elements are not the same as those in the model. A little further study shows that the bug is in our usage of the priority queue: we have failed to account for the fact that the assignment to *dest.weight* in *relax* (figure 1) *updates* the weights of nodes already in the queue. Because the queue is not sensitive to these updates, what it returns is no longer the smallest element in the queue. (Of course, these steps—of observing the discrepancy between the model and the phenomenon, then mapping it to actual understanding—require human ingenuity.)

On further reading, we trace the error to a subtle detail in the description of Dijkstra's algorithm in Cormen, et al.'s book [8, page 530]. The book permits the use of a binary heap (which is how we implemented the priority queue) for sparse graphs, but subsequently amends the pseudocode to say that the assignment to *dest.weight* must explicitly invoke a key-decrement operation. Our error, therefore, was not in the implementation of the heap, but in using the (faster) binary heap implementation without satisfying its (stronger) contract.

6 Reflections on the Example

While progressing through the example, we encounter several properties mentioned in the desiderata that make FrTime a good substrate for debugging. We review them here, point by point.

1. The DrScheme environment allows the user to keep and reuse abstractions across interactive sessions. For instance, to monitor the priority queue, we define procedures such as *not-in-order* and *convert-queue-to-list*. Such abstractions, which manipulate program data structures in a custom fashion, may be useful in finding and fixing similar bugs in the future. They can even become part of the program's distribution, assisting other users and developers. In general, debugging scripts can capture some of the *ontology* of the domain, which is embedded (but not always explicated) in the program.

2. We discover the bug by monitoring an invariant not explicitly represented in the program. Specifically, we keep a sorted list that mirrors the priority queue, and we observe that its behavior does not match the expectations of Dijkstra's algorithm. However, the list uses a linear time insertion procedure, which eliminates the performance benefit of the (logarithmic time) priority queue. Fortunately, by expressing this instrumentation as a debugging script, we cleanly separate it from the program's own code, and hence we incur the performance penalty only while debugging.

3. The interactive console of DrScheme, in which FrTime programs run, enables users to combine scripting with traditional interactive debugging. In the example, we first probe the priority queue at a coarse level, which

narrows the scope of the bug. We then extend our script to monitor the queue in greater detail. This ability to explore interactively saves the programmer from having to restart the program and manually recreate the conditions of the error.

4. The dataflow semantics of FrTime makes it well suited to act as a recipient of events and to keep models in a consistent state, even as the script is growing. During the execution of the Dijkstra solver, FrTime automatically propagates information from the variables *inserts* and *removes* to their dependents, the *violations* variable and the **set-running-e!** directive. Also, when we add the variable *model*, FrTime keeps it synchronized with *violations* without any change to the previous code.

5. The libraries of FrTime are rich enough to communicate with external entities. The programmer also has access to the programming constructs of DrScheme (higher-order functions, objects, modules, pattern-matching, etc.), which have rigorously defined semantics, in contrast to the ad-hoc constructs that populate many scripting languages. Further, since FrTime has access to all the libraries in DrScheme [15], it can generate visual displays and so on, as we will see in section 10.1.

7 Design

The design of MzTake contains four conceptual layers that arise naturally as a consequence of the goals set forth in the desiderata (Section 3).

First, we need abstractions that capture the essential functionality of a debugger. These are: observing a program's state, monitoring its control path, and controlling its execution. MzTake captures them as follows: **bind** retrieves values of program variables, **trace** installs trace points, and **set-running-e!** lets the user specify an event stream that starts and stops the program.

Second, we need a way to navigate the runtime data structures of the target program. For a Java debugger, this means providing a mechanism for enumerating fields and looking up their values.

Third, and most importantly, we need to be able to write scripts that serve as passive agents. Most general-purpose languages are designed to enable writing programs that control the world, starting with a "main" that controls the order of execution. In contrast, a debugging script has no "main": it cannot anticipate what events will happen in what order, and must instead faithfully follow the order of the target program's execution. Therefore we believe that a semantic distance between the scripting language and the kind of target language we are addressing is a necessary part of the solution.[4] Since the script's execution must be driven by the arrival of values from the program under observation, a dataflow language is a natural choice.

Once we have chosen a dataflow evaluation semantics, we must consider how broad the language must be. It is tempting to create a domain-specific debugging language that offers only a small number of primitives, such as

[4]Our work additionally introduces a *syntactic* difference when the target language is Java, but this can be papered over by a preprocessor.

<*debug-expr*>	::=	(**bind** (<*var*> . . .) <*expr*> . . .)
	|	(**trace** <*expr*> <*expr*>)
	|	(**set-running-e!** <*expr*>)
<*inspect-expr*>	::=	(*start-vm* <*expr*>)
	|	(**jclass** <*expr*> <*name*>)
	|	(**jloc** <*expr*> <*loc-expr*>)
	|	(**jdot** <*expr*> <*name*>)
<*loc-expr*>	::=	<*number*> | entry | exit
<*frtime-expr*>	::=	(*map-e* <*expr*> <*expr*>)
	|	(*filter-e* <*expr*> <*expr*>)
	|	(*merge-e* <*expr*> . . .)
	|	(*accum-b* <*expr*> <*expr*>)
	|	(*changes* <*expr*>)
	|	(*hold* <*expr*> <*expr*>)
	|	(*value-now* <*expr*>)
	|	*seconds*
	|	*key-strokes*
	|	(**lambda** (<*var*> . . .) <*expr*> . . .)
	|	(<*expr*> . . .)
	|	(**if** <*expr*> <*expr*> <*expr*>)
	|	... ; other Scheme expressions
<*expr*>	::=	<*debug-expr*>
	|	<*inspect-expr*>
	|	<*frtime-expr*>

Figure 9: MzTake Grammar

those we have introduced here. Unfortunately, once the script has gained control, it may need to perform arbitrary computational tasks, access libraries for input/output, and so forth. This constant growth of tasks makes it impractical to build and constantly extend this domain-specific language, and furthermore it calls into question the strategy of restricting it in the first place. In our work, we therefore avoid the domain-specific strategy, though we have tried to identify the essential elements of such a language as a guide to future language designers.

Having chosen a general-purpose strategy, we must still identify the right dataflow language. Our choice in this paper is informed by one more constraint imposed by debugging: the need to extend and modify the dataflow computation interactively without interrupting execution. Among dataflow languages, this form of dynamicity appears to be unique to FrTime.

We present the grammar of the MzTake language in Figure 9. The grammar is presented in layers, to mirror the above discussion. The first layer, represented by <*debug-expr*>, presents the most essential language primitives. The second layer, consisting of <*inspect-expr*> and <*loc-expr*>, represents primitives for obtaining information about the target program. The third layer describes the FrTime language.

8 Implementation

The examples we have seen so far describe a debugger for Java programs. However, the same principles of scriptable debugging should apply to most control-driven, call-by-value programming languages, with changes to take into account the syntactic and semantic peculiarities of each targeted language. To investigate the reusability of our ideas, we have implemented a version of MzTake for Scheme [21] also.

Not surprisingly, both the Java and Scheme versions share the design of the debugging constructs **trace**, **bind**, and **set-running-e!**. They differ in the operators they provide for accessing values in the language: because FrTime's data model is closer to Scheme's than to Java's, the Java version of the debugger requires a **jdot** operator to dereference values, but the Scheme version does not need the equivalent. Furthermore, because Java (mostly) names every major syntactic entity (such as classes and methods) whereas Scheme permits most values to be anonymous, the two flavors differ in the way they specify syntactic locations.

8.1 Java

The overall architecture of the Java debugger is shown in Figure 10.

On the left, we have the target Java program running on top of the virtual machine. The Java standard provides a language-independent debugging protocol called the Java Debug Wire Protocol (JDWP), designed to enable the construction of out-of-process debuggers. We have adapted a JDWP client implementation in Ruby [1] to DrScheme by compiling its machine-readable description of JDWP packets. We use this implementation to connect to the virtual machine over TCP/IP.

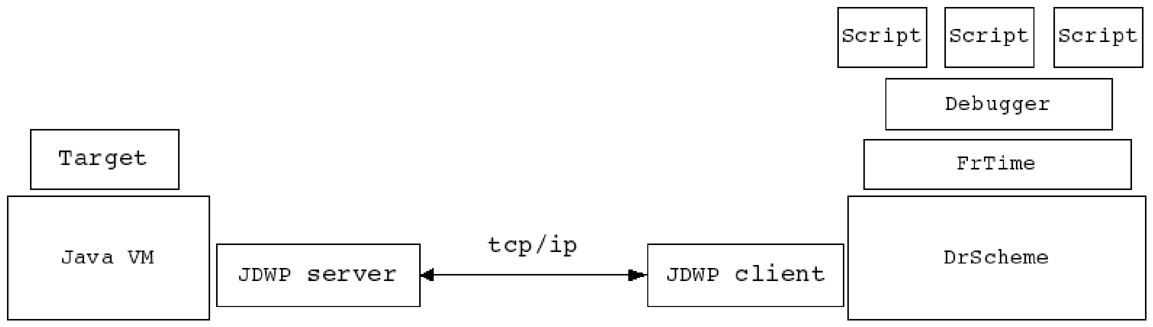


Figure 10: MzTake Architecture for Debugging Java

On the right of the figure, we have the stack of programming languages that we used to implement the debugger. FrTime is implemented on top of DrScheme, the debugging language is implemented on top of FrTime, and debugging scripts are themselves implemented in the debugging language.

The communication between the low-level debugger and the script proceeds in three stages. The first stage translates JDWP packets to a callback interface, the second dispatches these callbacks to their respective tracepoints, and the third translates them to FrTime event occurrences.

The second of these stages must handle subtleties introduced because the JDWP does not provide guarantees about the order in which messages arrive. For example, the following is a legal but troublesome sequence of messages. First, MzTake sends a message requesting a new tracepoint B. While MzTake waits for a reply, the target program reaches an existing tracepoint, A, generating an event that appears on the port before the virtual machine's reply to the request to install B. MzTake must either queue the trace at A while awaiting the acknowledgment of B or dispatch the A trace concurrently; it does the latter.

A trickier situation arises when a trace event at B appears even before the acknowledgment of installing that tracepoint. This is problematic because every trace event is tagged with a label that identifies which tracepoint generated it. This label is generated by the JDWP and communicated in the tracepoint installation acknowledgment. Therefore, until MzTake receives this acknowledgement, it cannot correctly interpret trace events labeled with a new tag. In this case, MzTake is forced to queue these events, and revisits the queue upon receipt of an acknowledgment.

We also need to translate the event callbacks into FrTime's event streams. Each usage of **trace** becomes associated with a callback. When the target reaches the traced location, its callback evaluates the **trace** expression's body and adds the result to FrTime's queue of pending events. It then posts on a semaphore to awaken the FrTime evaluator thread and waits. The event's value automatically propagates to all expressions that refer to the **trace** statement, directly or indirectly, in accordance with FrTime's dataflow semantics. When the FrTime dataflow graph reaches quiescence, the evaluator posts on a semaphore, which releases the callback thread and subsequently resumes the Java process. This provides synchronization between the debugging script and the debugged program. If the Java target program uses multiple threads, MzTake handles each event in a stop-the-world manner, to ensure that the script observes a consistent view of the program's state.

We found that the JDWP provides most of the functionality needed to build a scriptable debugger. Beyond implementing the packets and the dispatching as we mentioned above, we also needed to write two more components. The first was to duplicate Java's scoping rules in the implementation of **bind**: looking up *x* at a location first finds a local variable, if any, otherwise the field named *x* in the enclosing class, then in the super class, and so on. The second was to cache the results of JDWP queries pertaining to the fields of classes and the line numbers of methods, and flush the cache whenever the cached value might be invalidated; this is necessary to achieve both quick startup and acceptable runtime performance.

There are some other debugging events and inspection functions available in MzTake that we mentioned very briefly, or not at all, during the example. These include facilities for traversing the stack, enumerating local variables, and so on. There are also other events and functionality available through the JDWP that are not accessible in the debugger, such as class-loading and unloading events, static initializers, etc. What we have described so far is a conservative minimal extension of the programming language FrTime; it is easy to continue in the same vein to include support for the remaining events.

The inspection functions we provide pertain only to the data present in the target. We might like to reflect on the program's syntactic structure as well, for example to trace all assignments to a variable or all conditional statements. However, the JDWP does not provide support for such inspection, so we would need to build it on our own. In a sense, such capabilities are orthogonal to our work, since dataflow offers no new insight on processing of static syntax trees.

The quality of the JDWP implementation varied across virtual machines, and many versions were prone to crashes; we tested against the Sun JVM, the IBM JVM, and the Blackdown JVM, ultimately settling on the Sun implementation.

8.2 Scheme

The Scheme version employs source annotation. We instrument the Scheme program so that it mirrors the functionality of a process under the control of a debugger. The annotation mirrors the content of the lexical environment and introduces a procedure that determines when to invoke the debugger.

For example, suppose the original target program contains the expression

(**define** (*id x*) *x*)
(*id* 10)

The output of the annotator would be (approximately)

```
(define env empty)
(define (id x)
  (set! env (cons (list "x" x) env))
  (invoke-debugger 1 15 env)
  (begin0  ;; perform steps in order, then return value of the first expression
    x
    (set! env (rest env))))
(invoke-debugger 2 1 env)
(id 10)
```

When the annotated version executes, the *env* variable recreates the lexical environment. In particular, it tracks the *names* of variables in conjunction with their values, enabling inspection. The *invoke-debugger* procedure receives source location information (e.g., the arguments 2 and 1 refer to line two, column one). Each invocation of the procedure tests whether a tracepoint has been installed at that location and accordingly generates an event.

There are several important details glossed over by this simplified notion of annotation. We discuss each in turn:

thread-safety This annotation uses a mutable global variable for the environment. The actual implementation instead uses thread-local store.

tail-calls This annotation modifies the environment at the end of the procedure, thereby destroying tail-call behavior. The actual implementation uses *continuation-marks* [6], which are specifically designed to preserve tail-calls in annotations.

communication This annotation appears to invoke a procedure named *invoke-debugger* that resides in the program's namespace. Because FrTime runs atop the DrScheme virtual machine, the target Scheme program and the MzTake debugging environment share a common heap. Therefore, the annotation actually introduces a reference to the *value* of the debugging procedure, instead of referring to it by name.

The procedure *invoke-debugger* generates a FrTime event upon reaching a tracepoint, and then waits on a semaphore. From there, the evaluation of the script proceeds as in the Java case, since both implementations share the same FrTime evaluation engine. When the evaluation reaches quiescence, it releases the semaphore.

The implementation is available from

`http://www.cs.brown.edu/research/plt/software/mztake/`

8.3 Performance

We analyze the performance of the Dijkstra's algorithm monitor shown in figures 5 and 7. This example has a high breakpoint density (approximately 500 events per millisecond), so the time spent monitoring dominates the overall computation. In general, the impact of monitoring depends heavily on breakpoint density, and on the amount of processing performed by each breakpoint. All time measurements are for a 1.8GHz AMD Athlon XP processor running Sun's JVM version 1.4 for Linux.

We measure the running time of the the Dijkstra's algorithm monitor shown in figures 5 and 7, when it executes in the Java version of the debugger. Excluding the JVM startup time, it takes 3 minutes 42 seconds to monitor one million heap operations (either *add* or *extractMin*), which represents 2.217 milliseconds per operation. We partition this time into four parts: First, the virtual machine executes the call to either *add* or *extractMin* (0.002 milliseconds per operation). Second, the JDWP transmits the context information, FrTime decodes it, and FrTime schedules the recomputation (1.364 milliseconds per operation). Third, FrTime evaluates the script which monitors the partial correctness property, in figure 5 (0.851 milliseconds per operation).

According to these measurements, nearly one-third of the debugging time is devoted to JDWP encoding and decoding and to the context-switch. This is consistent with the penalty we might expect for using an out-of-process debugger. The time spent in FrTime can, of course, be arbitrary, depending on the complexity of the monitoring and debugging script.

In the Scheme implementation, the target and the debugger execute in the same process (while still preserving certain process-like abstractions [16]). As a result, whereas the Java implementation incurred a high context-switch cost, but no per-statement cost, the Scheme implementation incurs a small cost for each statement, but no operating system-level cost for switching contexts. Per operation, the annotation introduces a 0.126 milliseconds overhead. Thanks to the absence of a cross-process context-switch, dispatching an event costs 0.141 milliseconds per operation (compared with 1.3 milliseconds in the Java version of the debugger). The remaining times stay the same.

Obviously, MzTake is not yet efficient enough for intensive monitoring; we discuss this issue briefly in section 12. A two millisecond response time is, however, negligible when using MzTake interactively.

9 Controlling Program Execution

Debuggers not only inspect a program's values, but sometimes also control its execution. Some of the abstractions we defined in our running example were of the former kind (*not-in-order*, *convert-queue-to-list*). In contrast, we also defined a custom-purpose rule for deciding when to execute and when to pause, namely the function *nv*.

These *start-stop policies* represent a general pattern of debugger use. These policies can differ in subtle but important ways, especially when the same line has several breakpoints, each with its own callback. The start-stop policy used by most scripted debuggers consists of running the callbacks in order of their creation, until one of them requests a pause. Once this happens, the remaining breakpoints on the same line are not executed at all.

One might wonder, is this the right rule for all applications? In particular, preventing the execution of the subsequent callbacks creates a dependency between breakpoints (if the first breakpoint decides to suspend the execution, the second does not get to run at all). These dependencies are problematic if these breakpoints monitor implicit invariants

```
(define breakpoints (make-hash-table 'equal))

(define (break location callback)
  (let ([prev-breakpoint
         (if (hash-table-contains? breakpoints location)
             (hash-table-get breakpoints location)
             (trace location true))])

       (hash-table-put! breakpoints location
         (prev-breakpoint
          . ==> .
          (lambda (i) (if i (callback) false))))))

(define (resume)
  (set-running-e!
    (apply merge-e (hash-table-values breakpoints))))
```

Figure 11: A Typical Start-Stop Policy

```
(define breakpoints empty)

(define (break location callback)
  (set! breakpoints
    (cons (trace location (callback))
          breakpoints)))

(define (resume)
  (set-running-e!
    (apply merge-e breakpoints)))
```

Figure 12: A Different Start-Stop Policy

or implicit data structures, as we did during the example. During our debugging session, we created a mirror model of the queue so that it would elucidate the problem with the state of the real queue. In order to be of any debugging help, the model and the state must remain synchronized. If the event that detected the state violation prevented the execution of the event that updates the model, the program and model would cease to be synchronized. Worse, this would happen exactly when we need to look at the model, namely when we begin to explore the context of the violation.

By using a combination of first class events and **set-running-e!**, it is easy to define start-stop policies which are both custom-purpose and reusable. We implement the problematic start-stop policy just described with the code in figure 11. In the code, *breakpoints* is a hash table that maps locations to event streams. The *break* function sets or adds a breakpoint on a given line. The first time it is called on a given location, it installs a **trace** handler at that location, which simply sends the value true on the event stream each time the target program reaches that location. On subsequent invocations, it accumulates a cascade of events where each event is subordinate to the event that was in that location previously. When the execution of the target program reaches one of the locations, the script invokes each callback function in the cascade until the first one that returns false. The condition (**if** *i* . . .) ensures that the other callbacks are not called afterwards.

With MzTake, it is straightforward to define a different policy. Figure 12 shows the code for a break policy that executes all the breakpoints at one location before pausing the target program.

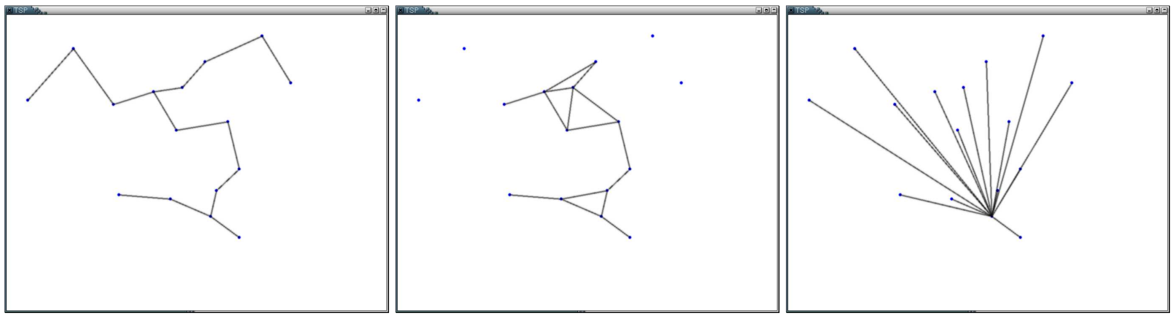


Figure 13: Spanning trees computed correctly (left), without detecting cycles (middle), and without sorting edges (right)

10 Additional Examples

In this section, we present some additional examples that further illustrate the power of our language.

10.1 Minimum Spanning Trees

Because MzTake has the full power of FrTime, users can take advantage of existing libraries to help them understand programs. For example, the FrTime animation library allows specification of time-varying images (i.e., image behaviors) that respond to events. Since MzTake generates events by tracing program execution, users can visualize program behavior by appropriately connecting these events to the animation library.

An intuitive visual representation can be an effective way of gaining insight into a program's (mis)behavior. Moreover, many programs lend themselves to natural visualizations. For example, we consider the problem of computing the Minimum Spanning Tree (MST) for a collection of points in the plane. (This example is based on the actual experience of one of the authors, in the context of writing a heuristic to solve the traveling-salesman problem.)

A simple greedy algorithm for the MST works by processing the edges in order of increasing length, taking each edge if and only if it does not introduce a cycle. Though the algorithm is straightforward, the programmer might forget to do something important, such as checking for cycles or first sorting the edges by length.

The programmer could write code to isolate the source of such errors, but a simple visualization of the program's output is much more telling. In Figure 13, we show visualizations of three versions of an MST program. On the left, we show the correct MST, in the middle, an edge set computed without cycle detection, and on the right, what happens if we forget to sort the edges.

In Figure 14, we show the debugging script that implements this visualization. Its salient elements are:

tree-start-event occurs each time the program begins computing a new MST, yielding an empty edge list

tree-edge-event occurs each time the algorithm takes a new edge, adding the new edge to the list

```
(define tree-start-event
  (trace ((tsp . jdot . mst) . jloc . entry)
         (bind () (lambda (prev) empty))))
(define tree-edge-event
  (trace ((tsp . jdot . mst) . jloc . 80)
         (bind (e)
           (lambda (prev)
             (cons (make-edge (e . jdot . v1)
                              (e . jdot . v2))
                   prev)))))
(define tree
  (accum-b (merge-e tree-start-event
                    tree-edge-event)
           empty))
(display-lines tree)
```

Figure 14: Recording MST Edges

tree builds a model of the tree by accumulating transformations from these event-streams, starting with an empty tree

display-lines displays the current tree

Though we have not shown the implementation of the MST algorithm, one important characteristic is that it does not maintain the set of edges it has taken: it only accumulates the *cost* of the edges and keeps track of which vertices are reachable from each other. In building an explicit model of the tree, our script highlights an important capability of our debugging system—it can capture information about the program's state that is not available from the program's own data structures. To implement the same functionality without a scriptable debugger, the user would need to amend the program to make it store this extra information.

10.2 A Statistical Profiler

Because our scripting language can easily monitor a program's execution, it should be relatively simple to construct a statistical profiler. Such a profiler uses a timer to periodically poll the program. Each time the timer discharges, the profiler records which procedure was executing and then re-starts the timer. The summary of this record provides an indication of the distribution of the program's execution time across the procedures.

MzTake provides a global time-varying value called **where**, which represents the current stack trace of the target

```
(define pings (make-hash-table 'equal))

((changes where)
 . ==> . (match-lambda [(line function context rest ...)
                         (hash-table-increment! pings (list function context))]
                        [_ (void)]))

(define ticks (changes (quotient milliseconds 50)))

(set-running-e! (merge-e (ticks . -=> . false)
                         (ticks . -=> . true)))
```

Figure 15: A Statistical Profiler

process. It is a list of symbolic locations starting with the current line and ending with the location of the *main* function. The value of **where** is updated any time the execution of the target is suspended, either by **trace** or by **set-running-e!**.[5]

Figure 15 uses **where** to implement a statistical profiler that records the top two stack frames at each poll. First, we instantiate a hash table to map stack contexts to their count. Next, each time the **where** behavior changes, we capture the current context and pattern-match on it using **match-lambda**. If the context contains at least a line, a function, and a caller function, we trim the context down to the function name and its caller and increment the count in the hash table. Then we bind *ticks* to a stream that sends an event every 50 milliseconds. Finally, we use **set-running-e!** to suspend the target at each tick. We want to resume the target soon after a pause, but how soon is soon enough? We want to leave just enough time so that the evaluation engine correctly updates the hash table before resuming the target, but no more. Recall that **set-running-e!** synchronizes with the evaluation of the script, so that it waits until all dependencies are fully recomputed before consuming the next event on its input stream. With that in mind, we use *merge-e* to create a stream containing two nearly-simultaneous events: the false tick is followed by a true tick immediately afterwards. The synchronization ensures that **set-running-e!** will not consume the true tick until the data flow consequences of the false ticks are completely computed.

This code only gathers profiling information. The script needs to eventually report this information to the user. There are two options: to wait until the program terminates (which the debugger indicates using an event), or to report it periodically based on clock ticks or some other condition. (The latter is especially useful when profiling a reactive program that does not terminate.) Both of these are easy to implement using FrTime's time-sensitive constructs.

[5]We also have another behavior *where/ss* (for *where with single stepping*) which updates at every step of the execution. This is useful for scripts that want to process the entire trace of the target. However, *where/ss* is disabled by default, for performance reasons.

11 Related Work

There are two main branches of research that relate to our work and from which we have drawn inspiration. We describe these in turn: first, programmable debugging, and second, program monitoring and instrumentation.

Dalek [26] is a scripted debugger built atop `gdb` that generates events corresponding to points in the program's execution. Each event is associated with a callback procedure that can, in turn, generate other events, thus simulating a dataflow style of evaluation. When the propagation stabilizes, Dalek resumes program execution.

MzTake has several important features not present in Dalek. A key difference that a user would notice is that we rely on FrTime to automatically construct the graph of dataflow dependencies, whereas in Dalek, the programmer must construct this manually. Dalek's events are not first-class values, so programmers must hard-wire events to scripts, and therefore cannot easily create reusable debugging operations such as *not-in-order*.

In Dalek, each event handler can suspend or resume the execution of the target program, but these can contradict each other. Dalek applies a fixed rule to arbitrate these conflicts, in contrast with the variety of start-stop rules we discussed in section 9. Indeed, using a stream as the guard expression highlights the power of using FrTime as the base language for the debugger, since we can reconstruct Dalek's policy in our debugger: the code shown in figure 11 is in fact Dalek's policy. This design addresses an important concern raised in an analysis of Dalek by Crawford, et al. [10].

The Acid debugger [29] provides the ability to respond to breakpoint commands and step commands with small programs written in a debugging script language very close to C. Deet [18] provides a scripting language based on Tcl/Tk along with a variety of the graphical facilities. Dispel [20] defines its own ad-hoc language. Generalized path expressions [5] specify break conditions as regular repressions applied to event traces. The regular expressions are augmented with predicate that can check for base-value relations. In these projects, the programmer must respond to events through callbacks, and there is no notion of a dataflow evaluation mechanism. Each retains the inspection and control mechanism of command-prompt debuggers.

DUEL [17] extends `gdb` with an interpreter for a language intended to be a superset of C. It provides several constructs, such as list comprehensions and generators, for inspecting large data structures interactively. However, it does not address how to control the target program or how to respond to events generated during the execution.

The Coca debugger by Ducassé [12] offers a conditional breakpoint language based on Prolog. Coca uses the backtracking evaluation mechanism of Prolog to identify potentially problematic control and data configurations during the execution, and brings these to the user's attention. As such, Prolog predicates serve as both the conditional breakpoint language and the data-matching language. However, since each predicate application happens in isolation from the other, there is no way to accumulate a model of the execution as it happens through time, such as constructing a trace history or building an explicit representation of an MST (as we have done in this paper).

Like Coca, on-the-fly query-based debugging [22, 23] enables users to interactively select heap objects. The objects are specified using a SQL-like language evaluated using an efficient on-line algorithm. It does not offer a sophisticated scripting mechanism. Like Coca, this approach does not support relating data between points in time.

Parasight [2] allows users to insert C code at tracepoint locations. The C code is compiled and inserted into the running target program's process in a way that has minimal performance impact. The inserted code must, however, adopt a callback-style to respond to events. While adapting the running program has performance benefits, it also complicates the process of using more expressive languages to perform monitoring and debugging (and indeed, Parasight does not tackle this issue at all, using the same language for both the target program and the scripts).

Alamo [19], like Parasight, instruments binary objects with in-process C code. While the scripts do not take the shape of callbacks, they must manually implement a programming pattern that simulates a coroutine (which is handled automatically in FrTime by the evaluation mechanism). The UFO debugger [4] extends Alamo with a rich pattern-matching syntax over events in terms of the target language's grammar. While MzTake offers a rich, general-purpose language for processing event-streams, UFO efficiently handles the special case of list comprehension followed by folding.

There are several projects for monitoring program execution, as Dias and Richardson's taxonomy describes [11]. Monitors differ from debuggers by virtue of not being interactive, and most do not provide scripting facilities. Instead, many of these systems have better explored the trade-offs between expressiveness, conciseness and efficiency in the specification of interesting events. MzTake simply relies on the powerful abstractions of FrTime to filter events, but at the cost of efficiency.

We have argued that the debugging code should remain outside the program's source code, to avoid complicating maintenance and introducing time- and space-complexity penalties. The debugging script is thus a classic "concern" that warrants separation from the core program. We could use aspect-like mechanisms [3] to express this separation. However, using them for our purposes would not be straightforward. Most implementations of aspect mechanisms rely on static compilation, which makes it impossible to change the set of debugging tasks on-the-fly. More importantly, most of them force the debugging script and main program to be in the same language, making it difficult to use more expressive languages for scripting. We therefore view these mechanisms as orthogonal to our work, and as possible routes for implementing our debugging language; in particular, MzTake would benefit from the "quantification" [14] provided by aspects.

Smith [27] proposes a declarative language for expressing equality constraints between the programmer's model and the execution trace. We can view this as an aspect-like system in which the aspects are not restricted to the original target language. Smith's language relies on a compiler to generate an instrumented program that maintains the model incrementally. Unfortunately, the compiler has not been implemented and, as the paper acknowledges, developing an implementation would not be easy.

Contracts [25] also capture invariants, but they too suffer from the need for static compilation. In addition, data structures sometimes obey a stronger contract in a specific context than they do normally. For instance, in our running example, priority heaps permit keys to change, which means there is no a priori order on a key's values. As we saw, however, Dijkstra's algorithm initializes keys to ∞ and decreases them monotonically; importantly, failure to do so results in an error. The topicality of the contract means it should not be associated with the priority heap in general.

Finally, unit testing frameworks provide a mechanism for checking that output from a function matches the expected answer. With respect to debugging, unit testing suffers from the same limitations as contracts. Namely, they operate statically and only along interface lines.

12 Conclusion and Future Work

We have presented the design and implementation of a scriptable, interactive debugger, and shown several instances of its application. The scripting language has sufficient library support to permit construction of a wide variety of applications including monitors and visualizations, and is powerful enough to make these concise to express. Along the way, we have demonstrated that MzTake can provide program-specific views of rich data, and can easily monitor implicit invariants.

MzTake does impose a burden on developers: to use it, they must learn a new language. This seems unavoidable for the reasons we have discussed in this paper. Indeed, more than the syntactic obstacles, the greater leap is to adjust to the dataflow programming model. Fortunately, the prevalence of event-driven software systems suggests a growing importance for dataflow languages in the future, so this investment has the potential to reap benefits in other domains as well.

The Scheme version of MzTake has been available for download since September 2004. The debugger has gathered some interest from the Scheme community. A number of people have downloaded it, and we have received enthusiastic feedback from several users.

In the future, we intend to use MzTake to specify temporal contracts on programs. It is typical for libraries to have restrictions on which function can be called at a given time, or in which order the functions can be called. For verification purposes, these temporal contracts are usually specified in terms of finite state machines. We expect that the data flow model can provide a more natural way of specifying them.

Another avenue of future work involves bringing the syntax and data model of the scripting language in the Java version of the debugger closer to Java. One possibility would be to use the Frappé system [9], a Java implementation of functional reactive programming that bears some similarity to FrTime.

We currently offer only weak ways of addressing program points. We would like to improve addressing in two ways. First, we intend to exploit the insights developed in aspect-oriented programming by adopting their *pointcut descriptors*. We would also like to avoid using line- and column-numbers, which are extremely brittle syntactically. It is possible that languages for querying semi-structured data can provide a foundation for this.

Acknowledgements

We thank the functional reactive programming group at Yale University, particularly John Peterson, Antony Courtney and Paul Hudak, for useful discussions. Thanks to Daniel Silva and John Clements who helped with the implementation. Thanks to the anonymous reviewers for helping improve the quality of our presentation. We also thank Manos Renieris, Philip Klein and Kathryn Doiron for their comments.

References

[1] The Ruby JDWP project. `http://rubyforge.org/projects/rubyjdwp/`.

[2] Ziya Aral and Ilya Gertner. High-level debugging in Parasight. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 151–162. ACM Press, 1988.

[3] Aspect oriented programming (article series). *Communications of the ACM*, 44(10), October 2001.

[4] Mikhail Auguston, Clinton Jeffery, and Scott Underwood. A framework for automatic debugging. In *Automated Software Engineering*, pages 217–222, 2002.

[5] Bernd Bruegge and Peter Hibbard. Generalized path expressions: A high level debugging mechanism. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging*, pages 34–44, 1983.

[6] John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

[7] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, 2006.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1997.

[9] Antony Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages*, pages 29–44. Springer-Verlag, March 2001.

[10] R. H. Crawford, R. A. Olsson, W. W. Ho, and C. E. Wee. Semantic issues in the design of languages for debugging. In *Proceedings of the International Conference on Computer Languages*, pages 252–261, 1992.

[11] Marcio de Sousa Dias and Debra J. Richardson. Issues on software monitoring. Technical report, ICS, 2002.

[12] Mireille Ducassé. Coca: an automated debugger for C. In *Proceedings of the 21st International Conference on Software Engineering*, pages 504–513, 1999.

[13] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the International Conference on Functional Programming*, pages 263–277, 1997.

[14] Robert Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns*, October 2000.

[15] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[16] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or, Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, September 1999.

[17] Michael Golan and David R. Hanson. DUEL - a very high-level debugging language. In *Proceedings of the USENIX Annual Technical Conference*, pages 107–118, Winter 1993.

[18] David R. Hanson and Jeffrey L. Kom. A simple and extensible graphical debugger. In *Proceedings of the USENIX Annual Technical Conference*, pages 183–174, 1997.

[19] Clinton Jeffery, Wenyi Zhou, Kevin Templer, and Michael Brazell. A lightweight architecture for program execution monitoring. In *SIGPLAN Notices*, volume 33, pages 67–74, 1998.

[20] Mark Scott Johnson. Dispel: A run-time debugging language. *Computer Languages*, 6:79–94, 1981.

[21] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), October 1998.

[22] Raimondas Lencevicius. On-the-fly query-based debugging with examples. In *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000.

[23] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engineering*, 10(1):39–74, 2003.

[24] Guillaume Marceau, Gregory H. Cooper, Shriram Krishnamurthi, and Steven P. Reiss. A dataflow language for scriptable debugging. In *IEEE International Conference on Automated Software Engineering*, 2004.

[25] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[26] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. Dalek: A GNU, improved programmable debugger. In *Proceedings of the Usenix Technical Conference*, pages 221–232, 1990.

[27] Douglas R. Smith. A generative approach to aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*, volume 3286, pages 39–54, 2004.

[28] Richard M. Stallman. *GDB Manual (The GNU Source-Level Debugger)*. Free Software Foundation, Cambridge, MA, third edition, January 1989.

[29] Phil Winterbottom. Acid, a debugger built from a language. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–222, January 1994.