Visualizing Program Execution Using User Abstractions

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912-1910
401-863-7641, FAX: 401-863-7657
spr@cs.brown.edu

Abstract

A practical system that uses visualization for understanding the execution of complex programs must offer the user views of the execution that are specific to the program being understood and the particular problem at hand without significantly slowing down or perturbing the system being examined. This paper describes a visualization data model and its implementation that accomplishes this. The model starts with program events that can be instrumented efficiently and with little overhead. It uses extended finite state automata to model program behaviors in terms of events. It builds time-varying data structures based on these automata. The data structures are then made available to various visualizations through a set of mappings that let the user dynamically control the visualization. The model and its implementation have been demonstrated on a range of specific understanding problems with a variety of different visualizations.

CR Categories: D.2.6 Graphical environments, D.2.5 Debugging aids.

Keywords: Dynamic software visualization, run-time monitoring, instrumentation.

1 Introduction

One of the principle goals of software visualization is to help programmers understand the intricacies of software systems. Many of the complexities of today's large, multithreaded, complex software systems involve their run time behaviors. To address these complexities we have long tried to develop dynamic software visualizations that can provide the appropriate insights.

Unfortunately, this is a difficult problem and there are essentially no visualizations today that are widely used that address the dynamic behavior of software systems. The goal of our research is to remedy this, to provide tools that can and will be used for understanding the dynamic behavior of software.

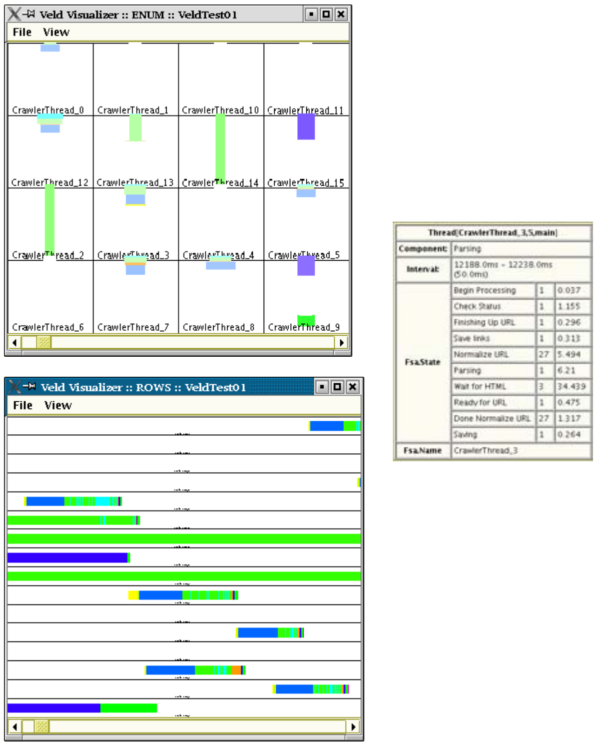
1.1 A Practical Approach

There are several issues that have to be addressed in order to make the visualization of software dynamics practical and desirable. These issues stem primarily from the fact that a worthwhile dynamic software visualization system has to address real systems and real problems.

A practical visualization system needs to address the types of software in which difficult understanding problems actually occur. This means that it has to be designed to handle large, multithreaded, often distributed, long-running computations and not just simple, single-user applications. The visualization system needs to handle these systems in real environments, which implies that the visualization must be done with minimal overhead and must be attachable to a running system. The visualization also must be closely correlated with the outside events that cause the dynamic behaviors of interest, which implies that the visualizations should be done in real time, with the user being able to go back and investigate any unusual situations in more detail after they occur.

A practical system also needs to address the wide range of actual problems that programmers face when trying to understand their software systems. Unfortunately, there is no single problem or even single class of problems that are commonly encountered here. Rather, almost every instance where the programmer needs to understand software behavior is a unique problem requiring a specialized solution and a visualization that is very problem and program specific. Generic solutions such as our prior work on JIVE [35] and JOVE [36], address generic problems and are often difficult to use in understanding the specific behaviors that programmers are actually interested in.

Based on this, we feel that a practical tool for visually understanding software behavior must meet certain criteria. First, it must have a model of what should be visualized. This needs to be problem and program specific, i.e. to be useful the visualization must address the specific problem of interest. Second, data based on this model needs to be gathered efficiently and with minimal program perturbation. Third, the system needs to provide a visualization that is useful for the particular data, one that not only displays the important information contained in the data, but also uses visual cues to highlight any unusual or unexpected data. This has to be done in a way that is intuitive for the programmer. Finally, the system needs to be easily configurable by the user. The developer needs to be able to quickly define the appropriate data model and visualization.



Thread[CrawlerThread_3,5,main]			
Component:	Parsing		
Interval:	12188.0ms - 12238.0ms (50.0ms)		
FsaState	Begin Processing	1	0.037
	Check Status	1	1.155
	Finishing Up URL	1	0.296
	Save links	1	0.313
	Normalize URL	27	5.494
	Parsing	1	6.21
	Wait for HTML	3	34.439
	Ready for URL	1	0.475
	Done Normalize URL	27	1.317
	Saving	1	0.264
FsaName	CrawlerThread_3		

FIGURE 1. Two visualizations of thread behavior in a web crawler in terms of the programmer. The colors reflect different program states for each of the threads. The first view reflects the amount of time each thread spends in each state during a short interface. The second view show the state changes over time. The text display on the right is an example tooltip showing what the mouse is pointing to in terms the programmer can understand.

1.2 Sample Problems

Determining the data model for understanding specific software behaviors requires knowledge of what types of behaviors programmers need understood. While such behaviors can be quite varied, we have chosen a set of sample problems that illustrate the types of issues that programmers actually need addressed based on our several years of recording what we would have liked to be able to visualize as we were programming. These provide motivation for our approach and form a basis for evaluating the resultant visualizations. Each of these problems is representative of a broader range of problems and reflects issues that we actually have faced in software development over the past few years. The specific problems include:

1. In a multithreaded web crawler, we want to have some idea how many threads to utilize. This requires understanding what the current threads are doing in terms of the program. Are they waiting for a web page, waiting for the robots.txt file to be processed for a site, parsing the HTML, processing text from the page, writing the various information to disk or the database, waiting for garbage collection, or just waiting for another page request. Here we need a visualization that shows the thread states in terms of the program such as that shown in Figure 1. Understanding thread behavior in terms of actual program states the general form of this problem.

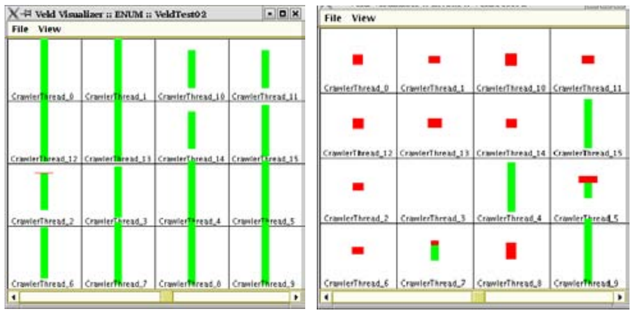


FIGURE 2. Two views showing time spent blocking while looking up HTML tags. The left view is the typical display showing that little if any time was spent blocking. The right view shows that there are times during the web crawler's execution where significant blocking occurs.

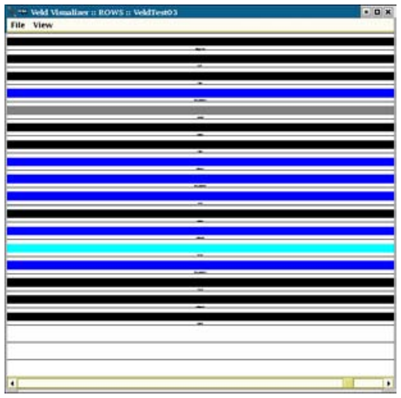


FIGURE 3. A visualization of the current states of the programmer's automata in a pinball program. The visualization shows a bar for each created automata and colors the bar by the current state at the current time. Tooltips can be used to determine the actual state and time of changes.

2. In the same crawler we noted that the default HTML parser uses a Hashtable for looking up HTML tags. We think that this might be a bottleneck in a multithreaded implementation (since it is synchronized) but are not sure. Can we tell? For this we created a visualization that showed when each of the threads was blocking on the hashtable as seen in Figure 2. From the visualization we determined that while most of the time this wasn't a problem, occasionally during the run it was an issue. This problem is a specialization of the first problem where we are interested in the behavior of a few particular states of the thread.

3. In a pinball program where students write the various automata that constitute the state of the simulated pinball machine, the students need to see the state of their different automata and how they change when different events occur in the game. A sample visualization of this is shown in Figure 3. The visualization shows each of the user's automata and their current state. Time is represented by the X-axis and state changes show up as changes in the colors of the bars. The general form of this problem involves understanding the behavior of user objects over time. This visualization is useful both to look at states of the objects and to look at the values associated with the objects.

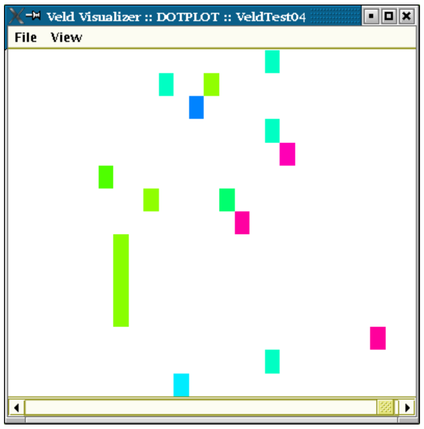


FIGURE 4. Dot-plot visualization showing dependencies between routines during flow analysis. The meaning of a dependency is inherent to the actual program. Color is used to show the age of the dependency. Tooltips can again be used to get more detailed information about a dependency.

4. In a system that does program flow analysis [39], we have a problem when a particular abstract routine is called but never returns. What we want to see is the dynamic nesting structure for each such routine showing what call or field access causes the routine to never return. Our previous approach involves generating and perusing multiple gigabyte trace files. Here we created a visualization that shows the dependency graph as a dot plot where the color of the dots depends on the age of the dependency. A sample result can be seen in Figure 4. This problem is representative of a set of problems where we want to visualize abstract program behavior (the flow analysis system never really builds the dependency graph), and illustrates the need for more complex data structures than simple time-varying values or states.
5. The specification of Iterators in Java says that *hasNext* should always be called before *next* is called. We want to check that our application follows this protocol. One resultant visualization uses simple boxes that show good iterators in green and bad ones in red, while another shows the location and state of iterators in terms of the source files. These can be seen in Figure 5. Understanding the behavior of protocols, either internal to an object or just exhibited by the program (or even a distributed system overall), is the logical extension of this type of visualization. Since protocols at all levels are important and often complex, this type of visualization can be very useful.

In this paper we describe a single data model that we use to address all these problems in a practical way. In particular we show how the model can be tied to minimal and dynamic instrumentation that can gather the necessary data efficiently and without perturbing the program and how the model can be used to drive a variety of different visualizations. We start by giving a summary of related work in the next section. We follow this by a description of the various parts of the model. This is followed by a discussion of the current implementation of the model and how visualizations should be defined. We conclude with our experiences and a description of the work remaining to be done to make this approach practical.

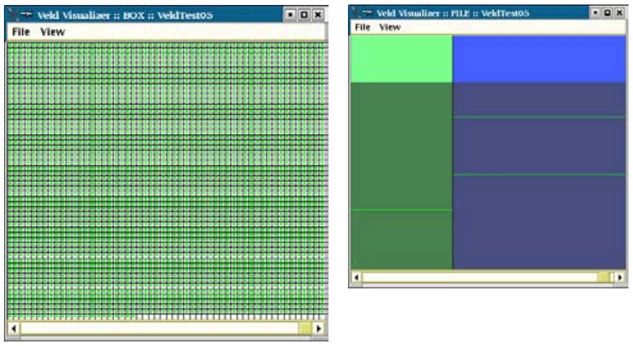

Veld Visualizer :: BOX :: VeldTest05
File View
Veld Visualizer :: FILE :: VeldTest05
File View

FIGURE 5. Two visualization showing the state of all the iterators used up to this point in a run. The first shows the state of each iterator, coloring one that is incorrectly used in red. The second shows the locations where iterators are used in the source files of the system and the corresponding states.

2 Related Work

Most visualizations of program execution today are not dynamic. These visualizations execute a program in a controlled environment while collecting trace data. They then analyze the trace data and create a visualization from the resultant analysis. This approach has several advantages. First the visualization tool can have access to a large amount of potential information by collecting a broad range of detailed trace data. Second, the visualizations can be more sophisticated since they don't have to be done in real time as the program executes. Third, once the run is complete, programmers can use the visualization at their leisure, spending time appropriate to understanding their problem. An industrial example of such a system is Jinsight [20-22]. Jinsight collects traces from Java programs and uses this data to provide a variety of displays that can be used to understand performance issues, class and object interactions, and memory utilization problems such as memory leaks. Among many other examples are [7,9-11,15,41,42] and our own work with Cacti [30] and BLOOM [31-33].

Having worked on and with such systems, we have come to the conclusion that they are not going to work for dealing with specific understanding questions involving the dynamics of modern software. There are several reasons for this. First, these systems tend to generate a significant amount of trace data. The cost of generating and storing this data means that the program under study executes at a significant slowdown. This makes the visualization difficult to use with today's systems which are interactive and long running. Second, because the trace files are large, the analyses that need to be done can also take a significant amount of time, meaning that the cost of just getting to the point of being able to see the visualization is high and is thus discouraging to potential users. Third, the systems typically model the whole run and show the user the result after execution over. This makes it very difficult for the user to correlate a particular external event with the visualization or even to remember what was going on at a point where the visualization might look interesting. Fourth, the collection of large amounts of trace data tends to perturb the behavior of the system, making problems involving timing or thread or process interaction diffi cultto reproduce during a trace run.

What is required and what is helpful are dynamic visualizations where the visualization runs along with the program and the program runs at close to its normal speed and without perturbation. Such systems require some compromise because they are limited both in terms of graphics and the data they can collect while still running the program at speed. However, the benefi tsof being able to correlate the visualization with what is currently going on, of being able to use the visualization on arbitrary systems, and of having the visualization overhead being relatively low, far outweigh the drawbacks. This is especially true if the visualization provides a history mechanism so that the user can go back (while the program is running or after it fi nishes)to look at a particular event or visualized anomaly in more detail.

There is a long history of such dynamic visualizations. The earliest computer-based visualizations of program execution showed the source code as it was executed. These visualizations typically highlighted a statement or line of code as the program was executing that line. These visualization were sometimes combined with other feedback information, for example execution totals or past history. Many of the early graphical programming environments featured some form of dynamic visualization along these lines. For example, our PECAN environment from the early 1980's outlined the current source statement with a box [23]. Other dynamically updated execution views provided by PECAN included a fl owchart view of the program and a view of the stack and the values of variables on it.

Along with PECAN there were algorithm animation systems such as Balsa [2-4], Tango [40], and others that included a view of the source code to highlight what the program was doing. These systems all worked because the programs under consideration were relatively small and execution time was not a primary concern. The algorithm animation systems generally showed other aspects of program execution as well. In particular, using either an event-based model or explicit code in the program, they displayed abstractions of the underlying data structures or the algorithm itself.

After PECAN we tried two different approaches to handling more realistic programs. First, the GARDEN system attempted to do it using abstraction [24,26]. GARDEN was a programming system that let the user defi ne,integrate, and use new visual languages. Each language had a graphical syntax and an execution semantics defi nedin terms of other languages or GARDEN primitives. Programs were represented by objects that could be executed directly. GARDEN provided the hooks to automatically highlight execution within the visual displays of a program.

Our second approach was in the FIELD system. Here we attempted to provide visualization of full-sized C (and later Pascal, Object Pascal, and C++) systems [27,29]. FIELD included source level views that updated whenever the debugger stopped execution. Moreover, it supported automatic single stepping so that the user could view the program execution in the editor. Similar views can be found in [17].

In addition to visualizing the source, FIELD provided visualizations of user data structures that were updated dynamically as the program executed. The user was given control over when to update the structure to keep performance reasonable. This is similar to the displays provided by other tools [1,19] and later commercial environments from SGI and Sun. FIELD also let the user customize the data structure displays [25].

Since source lines are too coarse a representation to show dynamic execution of large systems, visualizations soon moved to more abstract forms. The idea here is to take a higher level view of the program and then to show the execution dynamically in terms of that view. FIELD provided several such views including displaying execution in call graphs and a class browser, and providing specialized views that showed performance, I/O behavior in terms of fi les,and memory behavior. Other systems that have provided similar performance visualizations include IBM's PV [14] the visualizations that accompany MPI, or the visualizations incorporated in Sun's programming environment.

Extending this concept, out next dynamic visualization system, JIVE, combined several abstractions into one visualization [35,37]. One of these abstractions provides a view of execution in terms of classes or packages, two others provide views of thread behavior in terms of execution states. A fourth provided a estimation of the program phase [38]. JIVE worked by breaking the execution into 20 millisecond intervals, accumulating data internally during the intervals, and then visualizing the summary data in a separate concurrent process. JIVE had low enough overhead to be used on arbitrary Java programs

Going beyond JIVE, we built a second visualizer, JOVE, that maintains a more complex model of program behavior [36]. It again looks at the program in terms of small intervals. For each interval it keeps track of how many times each basic block is executed by each thread. The summary information is then kept over the history of the run. Using sophisticated instrumentation, we were able to display the resultant summary data in real time without signifi cantly afecting program behavior.

The problem with both JIVE and JOVE and other current dynamic visualizers is that they provide generic solutions that do not fi tthe specifi cproblems that programmers have. We wanted to build on previous work by ourselves [32-34] and systems such as EVolve [41,42] that attempt to let the programmer craft visualizations to fi ttheir problems. However we wanted to do so in a dynamic rather than a static or post-mortem environment.

3 Visualization Data Model

Doing high-quality, program-specifi cvisualization in a dynamic environment puts strong requirements on data collection, data analysis, and visualization. In particular, it means that

- All aspects of the system must be very effi cient.The instrumentation must not signifi cantlyslow down the program. The amount of data collected must be small enough to move rapidly from the executing program to the visualizer. The data analysis must be done in real time. The visualization itself must provide dynamic frames at a smooth, real-time frame rate. Our experience has shown that typical data collection techniques for Java (e.g. using JVMTI, JVMPI, or JVMDI) slow the program down by a factor of at least ten and introduce unnecessary thread synchronizations.

- Instrumentation should take great pains to not perturb program execution. One has to avoid instrumentation strategies result in inadvertent thread synchronizations or significant slowdowns. Similarly, visualizations that are too costly to compute or that can't be done in real time will affect the correlation between the execution and the visualization.
- The data that can be collected has to be general enough to reflect a wide range of different problems and to provide appropriate abstract views of program execution. The information needed to address the examples at the start of this paper include determining where threads change states, when a thread is waiting on a particular monitor, how the program implicitly builds a dependency graph, the state of internal classes that happen to reflect automata, and the behavior of class instances representing iterators as determined by call sequences.
- The visualization must be determined by an analysis of this data. This analysis can be as simple as the current state for each thread. It can also be history dependent, for example, in the behavior of Iterators where the sequence of calls determines what is legal or illegal. It can also be simply implicit, for example as seen in the dependency graph that the flow analyzer never really builds but that can be inferred from various behaviors.

Our approach is driven by these requirements. We start by using parameterized events defined in abstract terms over the program. These are used to specify what parts of the execution are relevant to the problem at hand. These events are used to define and drive finite state automata that are used to reflect the time-dependent behavior that should be visualized. These automata are then used to build time-aware data structures by defining data-structure modifying actions as events on their transitions. The data structures are then used as the basis for a variety of different visualizations. These various components of the model are described in detail in the next sections.

3.1 Parameterized Program Events

The starting point for a visualization data model for execution needs to be rooted in the execution itself. Previous work, starting with Balsa [5] has demonstrated the utility of using "interesting" events as the basis for describing program behavior. While algorithm animation systems such as Balsa did manual instrumentation at the source level, we needed to do more automatic instrumentation and to work with real programs where we don't always have actual source code. Here we noted that the problem of identifying interesting aspects of the execution is essentially the same as that faced by aspect-oriented programming.

Aspect/J provides a language that lets the program specify points in the program where aspects should be attached [12,13]. The implementation of this involves identifying the corresponding bytecode and inserting appropriate instrumentation calls. Our system works much the same and could even make use of the Aspect/J syntax and instrumentation packages. However, because we wanted to correlate the dynamic patching and unpatching of distributed systems, we started with a small patcher of our own on top of JikesBT [16].

The set of program events, again similar to Aspect/J, includes calls, returns, method entry and exit, allocations, field sets, throwing and catching exceptions, and setting and releasing monitors. Each of these events can be restricted based on location or type information. Moreover, each is parameterized with appropriate program data. For example, a call event can be restricted to calls of a given method or calls from a given method, and can have as parameters the current object, the current thread, any of the call parameters, or any data directly accessible from one of these items.

3.2 Event Automata

The program events drive extended finite state automata. These automata represent an abstraction of the program behavior that is of interest to the visualization. For example, if the programmer is interested in the behavior of a particular protocol in the application, the automata would describe the states of the protocol including information as to which state transitions were valid. Events are used here both to define the automata and to cause the automata to change states appropriately.

Many of the interesting problems for visualization involve multiple instances to be visualized. For example, we want to understand the behavior of each thread in the web crawler separately and we want to independently monitor the state of each Iterator in the application. This means that we need to maintain multiple automata and relate program events to the appropriate automata. This is done using the event parameters.

Each relevant parameter for an event has an associated match setting. This setting indicates whether the event should create a new automaton with the parameter value as the key or whether it should find an existing automaton using the parameter value. The system supports automaton defined using multiple parameters coming from separate events to support understanding behaviors based on sets of objects.

The automata model that is used is extended to allow each automaton to have a set of local variables and to allow conditions on the arcs based on these variables. This makes it relatively easy to define automata that can count (to ensure for example that the number of pushes equals the number of pops) and to describe most interesting behaviors in a simple manner.

The event automata are created and driven from the events. The system associates a set of event actions with each event. These actions can create automata with a given key, look up automata using a parameter as key (if the problem requires multiple lookup events, this returns a set of automata rather than the actual automaton), set parameters in the current automaton, and activate the

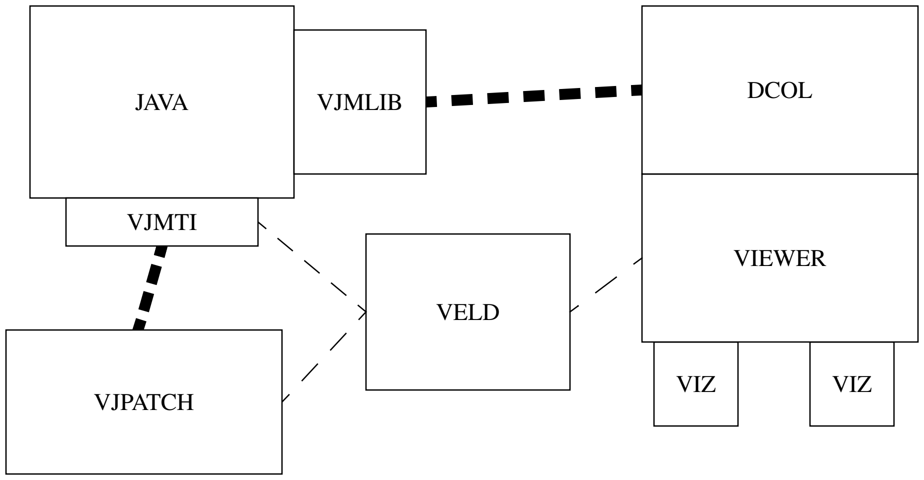


FIGURE 6. An overview of the visualization architecture.

current automaton. The event is also used to cause transitions once the current automaton is active.

Our instrumentation system can build and maintain these automata either in the monitored process or in the visualization process. The former is generally more effi cientand allows the system to process and summarize the data locally before sending it to the visualizer. It is used if only summary data is required. The latter is used when more detailed information such as every state transition is going to be visualized or if the cost of creating and maintaining automata over the program run would be excessive.

3.3 Data Structures

The event automata provide a tractable abstraction of program behavior. In many cases what the programmer wants visualized is a direct representation of this behavior, and the visualization could be derived directly from the automata. However, other instances involve more detailed analysis of the behavior and most visualizations require time-based data.

To accommodate these needs our model creates time-aware data structures that refl ect more directly what should be visualized. We provide basic data structures consisting of entities and fi elds and a specialized implementation of a graph with corresponding graph operations.

For the basic data structures, each fi eldcan be either fi xed or time varying. Access to the data structure from the visualizers specifi eseither a particular time or a time interval. For a particular time, the implementation returns the fi eldvalue at that time. For a time interval, the implementation will return the set of values that the fi eldheld during that interval along with information as to how many times each value was set and how much time during the interval the fi eldheld the value. The implementation can also return, when desired, the sequence of value changes and times for the particular fi eld.Graphs are treated like basic data structures with entities refl ecting nodes and arcs that can have associated values.

The data structure is created and maintained by defi ningactions on the transitions of the event automata. These actions let the event create entities, fi ndentities based on parameter values, set fi elds,and add or remove graph nodes and arcs. While these actions can be defi nedexplicitly, the implementation provides automatic defi nitionsfor the common case where the data structure is a direct refl ectionof the event automata, that is where there is one entity for each automata and that entity has fi eldsfor the current state and each of the associated variables.

3.4 Visualization Mappings

The data structures provide access to the data that is relevant to the problem for which the programmer wants to use visualization. However, it is rarely the case that one can determine a priori how that data should be displayed to make best use of the visualization technology. Different fi eldsmight need to be stressed at different times and in different ways. Value ranges might need to be compressed or expanded to best highlight the underlying data. With time-varying data, the number of changes may be more relevant than the time spent with each value or visa versa. The visualization might also only want to view values that meet a certain criteria, for example delays greater that a given threshold or Iterators derived from methods in a particular package.

To accommodate this and to make the visualization easier to code, access to the data structures is done indirectly through a set of mappings. Each mapping specifi esthe fi eldand entity that is being addressed, whether the program wants the actual values, counts, or times, a value fi lter, and a scaling function. Each graphical property of the visualization is associated with its own mapping. Moreover, the system lets the mappings be changed dynamically to give the programmer control of the visualization.

4 Implementation Framework

The visualization system is built as a suite of separate components that communicate via sockets and an updated version of the FIELD message server [28] as seen in Figure 6.

```
<VELD>
    <VISUALIZATION NAME='ENUM' CLASS='edu.brown.cs.veld.viz.VizEnum'>
        <PARAMETER NAME='ShowText' LABEL='Display Text' TYPE='BOOL' DEFAULT='true' />
        <PARAMETER NAME='SortItems' LABEL='Sort Items' TYPE='BOOL' DEFAULT='true' />
        <ENTITY NAME='BASE' />
        <STATISTIC NAME='HUE' LABEL='Hue' TYPE='ENUM' ENTITY='BASE' COLOR='true' />
        <STATISTIC NAME='SAT' LABEL='Saturation' TYPE='VALUE' ENTITY='BASE' />
        <STATISTIC NAME='INT' LABEL='Intensity' TYPE='VALUE' ENTITY='BASE' />
        <STATISTIC NAME='HEIGHT' LABEL='Height' TYPE='VALUE' ENTITY='BASE' />
        <STATISTIC NAME='WIDTH' LABEL='Width' TYPE='VALUE' ENTITY='BASE' />
        <STATISTIC NAME='TEXTURE' LABEL='Texture' TYPE='VALUE' ENTITY='BASE' />
        <STATISTIC NAME='LABEL' LABEL='Label' TYPE='LABEL' ENTITY='BASE' />
    </VISUALIZATION>
</VELD>
```

FIGURE 7. The visualization description for enumerations. This first defines two visualization parameters that will be made available to th user. It then describes the entities expected by the visualization. This visualization works with one type of entity. Finally, it describes the various visual properties that the user can set for the visualization. These are the properties that can be associated with the visualization dat using the visualization mappings.

The central box labeled VELD acts as the visualization controller. It is responsible for creating the various system components and provides the facilities needed to let the user define and control the visualization. It also maintains the visualization data model, making it accessible to the various components.

The user's application runs as a normal Java process as seen in the upper left of the figure. We provide two additions to that process. The first, labeled VJMTI, uses JVMTI [18] to interface between the Java virtual machine and the rest of the visualization system. It provides several functions. First, it traps class loading so that classes can be instrumented. Second, it provides the ability to dynamically exchange instrumented and uninstrumented classes, thereby letting the system attach and detach visualizations from the application. Third, it lets the visualization controller control data collection by enabling or disabling collection and setting the accumulation interval dynamically. Fourth, it lets the controller query the set of available processes that can be visualized. Communication between the controller and VJMTI is through the message server. This permits multiple virtual machines running on the same or different machines to be used in the same visualization.

The second addition to the user process, labeled VJMLIB in the figure, is the library called by the instrumented code. This library can operate in one of three modes depending on the requirements of the visualization. In one mode it passes all events to the data collector, DCOL, using a socket. In the second mode it maintains all the event automata and passes the data structure manipulation actions to the data collector. This mode, which is typically used, generally results in less communication but slightly more overhead. This overhead is reduced further by having VELD generate visualization-specific code for maintaining the event automata. The third mode is used when only summary data is desired. Here the library collects statistics on field transitions for short execution intervals, typically 30ms, and only sends the summary data to the collector. This library is crafted so that events are collected without any thread synchronization, so that messages are buffered and sent to the data collector periodically in batches, and so that data processing and communication can overlap. This is done by creating a separate message area for each thread, by triple buffering these areas within the library, and by having a separate thread in charge of communications.

The data collector is primarily concerned with providing interval-based access to the user data structures. It maintains event automata if necessary. For each time-varying field it keeps the set of transitions or summary data that has been sent. The visualization can then request data for a given interval. When it gets such a request, the data collector computes the data for the given interval based on the raw data it had collected. Data for the computed interval is cached to minimize recomputation.

Access to the data collector is through a view controller labeled VIEWER in the diagram. This package is responsible for handling the visualization mapping portion of the model. In addition, it controls the various visualizations and handles time synchronization of different views. It is also responsible for providing the user interface for visualization, letting the user view the history of the execution or the current state, letting the user change the visualization mappings, and letting the user change the parameters that characterize each of the visualizations.

The visualizations themselves are designed as independent plugins for the view controller. This makes it relatively straightforward to create new visualizations and lets the system be used with a wide variety of different visualization strategies. The current set of visualizations include boxes (shown on the left in Figure 5), Dot plots (Figure 4) [6], Enumerations (top of Figure 1 and Figure 2), File-based displays akin to SeeSoft (Figure 5 on the right) [8], and time bars (bottom of Figure 1 and Figure 3).

The different visualizations implement a simple interface defined by the view controller and are described in an XML file. Each visualization describes its own data model in terms of the entities that it expects, the set of graphical properties that can be drawn from those entities, and the set of parameters that characterize the visualizations. For example, Figure 7 shows the description of the enumeration visualization.

```
<VELDVIZ NAME='VeldTest02'>
  <DESCRIPTION>Track the time spend getting html tags to check for locks</DESCRIPTION>
  <EVENTS>
    <EVENT ID='E0' NAME='Create Thread' TRIGGER='1'
          TYPE='ENTER'
          METHOD='edu.brown.cs.cs032.crawler.crawl.CrawlThread.run'>
      <MATCH ID='THIS' PARAM='P1' TYPE='NEW' NAME='1' />
    </EVENT>
    <EVENT ID='E1' NAME='Lookup HTML'
          TYPE='ENTER'
          METHOD='javax.swing.text.html.HTML.getTag'>
      <MATCH ID='FROMTHREAD' PARAM='P1' TYPE='MATCH' />
    </EVENT>
    <EVENT ID='E2' NAME='Lookup HTML'
          TYPE='EXIT'
          METHOD='javax.swing.text.html.HTML.getTag'>
      <MATCH ID='FROMTHREAD' PARAM='P1' TYPE='MATCH' />
    </EVENT>
  </EVENTS>
  <AUTOMATA START='S0'>
    <STATE ID='S0' NAME='Start'>
      <ON EVENT='E0' GOTO='S1' />
    </STATE>
    <STATE ID='S1' NAME='Running'>
      <ON EVENT='E1' GOTO='S2' />
      <ON EVENT='E2' GOTO='S1' />
    </STATE>
    <STATE ID='S2' NAME='Lookup HTML'>
      <ON EVENT='E1' GOTO='S2' />
      <ON EVENT='E2' GOTO='S1' />
    </STATE>
  </AUTOMATA>
  <DATA TYPE='AUTOMATA'>
    <SHOW WHAT='STATE_TIMES' />
  </DATA>
  <VIEW NAME="view1" TYPE='ENUM'>
    <ENTITY NAME='BASE' VALUE='Fsa' />
    <MAP VIEW='HUE' ENTITY='BASE' FIELD='State' PROP='ENUM'>
      <COLOR RGB='0x00ffffff' />
      <COLOR RGB='0x0000ff00' />
      <COLOR RGB='0x00ff0000' />
    </MAP>
    <MAP VIEW='WIDTH' ENTITY='BASE' FIELD='State' PROP='COUNT' />
    <MAP VIEW='HEIGHT' ENTITY='BASE' FIELD='State' PROP='TIME' />
    <MAP VIEW='SAT' ENTITY='BASE' PROP='HIGH' />
    <MAP VIEW='INT' ENTITY='BASE' PROP='LOW' />
    <MAP VIEW='TEXTURE' ENTITY='BASE' PROP='NONE' />
    <MAP VIEW='LABEL' ENTITY='BASE' FIELD='Name' />
  </VIEW>
</VELDVIZ>
```

FIGURE 8. Sample visualization description file. The file describes the events, the event automata, the data model, and any views that are associated with the visualization along with their current visualization mappings.

5 Defining Visualizations

The overall system maintains a complete model of the desired visualization and shares that model with all the components. This model can be saved so that the same visualization can be recreated for the same or for a different application. The model is maintained in memory by the visualization controller, VELD, and is stored and shared between tools using an XML file. An example of this file can be seen in Figure 8. The file provides a description of the visualization, a listing and description of the events that need to be instrumented in the application, a description of the automata, a description of the data structures, and a description of the views associated with the visualization. In this case it uses the default data model that shows the states of the event automata. In addition, the model contains information about how the data structure elements are to be used. In this case, the model specifies that the viewer is only interested in the time spent in each state of the event automata. Additional specification can provide information on what program to run and what classes to instrument or ignore.

Currently visualizations are constructed by creating this file in an editor. We are working on several tools that will automate this process and have designed the overall system to facilitate this.

First we are working on a tool that uses wizards to let the programmer quickly characterize their visualization and then provide the small amount of information needed by the wizard. In conjunction with this, we are developing a tool that lets the programmer edit each of the model components separately using an appropriate framework. For example, events can be edited by pointing to the appropriate source locations, a graphical editor is provided for defining automata, and views can be selected and parameterized through simple dialog boxes.

The model contains enough information so that once the data structure model is provided, it can match that model and the properties of interest with the set of available visualizations so that visualizations could be selected automatically.

6 Experience and Future Work

The model described in this paper is necessary for a practical dynamic visualization system. Any system that attempts to do large-scale dynamic visualization without addressing all the issues that the model addresses is not going to be able to provide the necessary range of visualizations or the necessary flexibility in adapting visualizations to meet the real understanding problems that need to be addressed.

The key components of the model include starting from program events, modeling program behavior, particularly protocols, using extended finite automata, providing a suite of data structures to contain the information to be visualized, and offering access to this data based on time intervals and through mappings that can provide the most appropriate information for visualization.

We have used our implementation to build visualizations for the problems listed earlier (as seen in the corresponding figures). In each case, we are able to run the original application without noticeable slowdown and to see the resultant visualizations in real time. The actual slowdown depends on a variety of factors including the number of items being monitored, the frequency of changes to the monitored items, the type of summary data being sent, the frequency of updates, whether the program is cpu-bound or IO-bound, and the ability of the computer to do the necessary graphics if the visualization is run on the same machine. In almost all cases we tried, the program seemed to run at normal speeds. In the worst case situation, with a CPU bound program and high-intensive monitoring (looking at all Iterators), the slowdown was less than a factor of 2.

Our experience with the prototype tool has been positive. We have been able to correlate changes in the visualization to events in the program. We have used the visualizations to understand program behavior. For example, the visualizations of the web crawler showed that much of the delay that we were experiencing was while waiting to get the robots.txt information rather than waiting for web pages. This was a result that did not show up with the earlier generic visualizations. Overall, we have shown through example problems and the resultant visualizations that the model and our implementation of it is practical and has broad applicability.

The model is necessary for practical dynamic visualization, it is not sufficient. A truly practical visualization system needs to go beyond the underlying framework and provide a user interface that is both intuitive and easy to use. Without such an interface, the work required to define and use a visualization outweighs the benefits of using visualization rather than using traditional approaches. There are two parts to the interface. The first involves letting the programmer describe the problem so that a visualization model can be built. The second involves letting the programmer manipulate and understand the visualization itself. We are currently working on these two aspects of the system, confident that because we have the right underlying framework we will be able to create a tool that will actually be used. Our approach here should allow rapid and continual experimentation with dynamic visualization of the arbitrary Java applications.

7 Acknowledgements

This work was done with support from the National Science Foundation through grants CCR021897 and ACI9982266.

8 References

1. David B. Baskerville, “Graphic presentation of data structures in the DBX debugger,” UC Berkeley UCB/CSD 86/260 (1985).

2. John Bazik, Roberto Tamassia, Steven P. Reiss, and Andries van Dam, “Software visualization in teaching at Brown University,” pp. 383-398 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, MIT Press (1998).

3. Marc H. Brown and Steven P. Reiss, “Debugging in the BALSA-PECAN integrated environment,” ACM SIGPLAN-SIGSOFT Symposium on Debugging (1983).

4. Marc H. Brown and Robert Sedgewick, “A system for algorithm animation,” *Computer Graphics* Vol. **18**(3) pp. 177-186 (July 1984).

5. Marc H. Brown and Robert Sedgewick, “Interesting Events,” pp. 155-172 in *Software Visualization*: *Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, MIT Press (1998).

6. Kenneth W. Church and Jonathan I. Helfman, “Dotplot: a program for exploring self-similarity in millions of lines for text and code,” *Journal of Computational and Graphical Statistics* Vol. **2** pp. 153-174 (1993).

7. Mikhali Dmitriev, “Design of JFluid: A profiling technology and tool based on dynamic bytecode instrumentation,” *Sun Microsystems Report TR_2003-125*, (November 2003).

8. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr., “Seesoft - a tool for visualizing software,” AT&T Bell Laboratories (1991).

9. Paul Gestwicki and Bharat Jayaraman, “Methodology and Architecture of JIVE,” *SoftVis 2005*, pp. 95-104 (May 2005).

10. Dean Jerding, John T. Stasko, and Thomas Ball, “Visualizing interactions in program executions,” *Proc 19th Intl. Conf. on Software Engineering*, pp. 360-370 (May 1997).

11. Dean F. Jerding, “Visualizing patterns in the execution of object-oriented programs,” pp. 47-48 in *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*, (1996).

12. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An Overview of AspectJ," in *European Conference on Object-Oriented Programming*, (2001).

13. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming," in *European Conference on Object-Oriented Programming*, (jun 1997).

14. Doug Kimelman, Bryan Rosenburg, and Tova Roth, "Visualization of dynamics in real world software systems," pp. 293-314 in *Software Visualization: Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

15. Eileen Kraemer, "Visualizing concurrent programs," pp. 237-256 in *Software Visualization: Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

16. Chris Laffra, Doug Lorch, Dave Streeter, Frank Tip, and John Field, "What is Jikes Bytecode Toolkit," *http://www.alphaworks.ibm.com/tech/jikesbt*, (March 2000).

17. Henry Lieberman and Christopher Fry, "Bridging the gap between code and behavior in programming," *CHI '95*, pp. 480-486 (April 1995).

18. Sun Microsystems, "JVM Tool Interface," *http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html*, (2004).

19. Brad A. Myers, "Displaying data structures for interactive debugging," Xerox csl-80-7 (June 1980).

20. Wim De Pauw, Doug Kimelman, and John Vlissides, "Visualizing object- oriented software execution," pp. 329-346 in *Software Visualization: Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price,MIT Press (1998).

21. Wim De Pauw and Gary Sevitsky, "Visualizing reference patterns for solving memory leaks in Java," in *Proceedings of the ECOOP '99 European Conference on Object-oriented Programming*, (1999).

22. Wim De Pauw, Nick Mitchell, Martin Robillard, Gary Sevitsky, and Harini Srinivasan, "Drive-by analysis of running programs," *Proc. ICSE Workshop of Software Visualization*, (May 2001).

23. Steven P. Reiss, "PECAN: program development systems that support multiple views," *IEEE Trans. Soft. Eng.* Vol. **SE-11** pp. 276-284 (March 1985).

24. Steven P. Reiss, Eric J. Golin, and Robert V. Rubin, "Prototyping visual languages with the GARDEN system," *Proc. IEEE Symp. on Visual Languages*, (June 1986).

25. Steven P. Reiss and Joseph N. Pato, "Displaying program and data structures," *Proc. 20th Hawaii Intl. Conf. System Sciences*, (January 1987).

26. Steven P. Reiss, "Working in the Garden environment for conceptual programming," *IEEE Software* Vol. **4**(6) pp. 16-27 (November 1987).

27. Steven P. Reiss, "Interacting with the FIELD environment," *Software Practice and Experience* Vol. **20**(S1) pp. 89-115 (June 1990).

28. Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software* Vol. **7**(4) pp. 57-67 (July 1990).

29. Steven P. Reiss, *FIELD: A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).

30. Steven P. Reiss, "An engine for the 3D visualization of program information," *Journal of Visual Languages*, (December, 1995).

31. Steven P. Reiss and Manos Renieris, "Encoding program executions," *Proc ICSE 2001*, (May 2001).

32. Steven P. Reiss, "Bee/Hive: a software visualization backend," *IEEE Workshop on Software Visualization*, (May 2001).

33. Steven P. Reiss, "An overview of BLOOM," *PASTE '01*, (June 2001).

34. Steven P. Reiss, "A visual query language for software visualization," *IEEE 2002 Symposium on Human Centric Computing Languages and Environments*, pp. 80-82 (September 2002).

35. Steven P. Reiss, "JIVE: visualizing Java in action," *Proc. ICSE 2003*, pp. 820-821 (May 2003).

36. Steven P. Reiss and Manos Renieris, "JOVE: Java as it happens," *Proc. SoftVis '05*, pp. 115-124 (May 2005).

37. Steven P. Reiss, "Efficient monitoring and display of thread state in Java," *IWPC 2005*, pp. 247-256 (May 2005).

38. Steven P. Reiss, "Dynamic detection and visualization of software phases," *Proc. Third International Workshop on Dynamic Analysis*, (May 2005).

39. Steven P. Reiss, "Checking event-based specifications in Java systems," *Proc. SoftMC 2005*, (July 2005).

40. John T. Stasko, "TANGO: a framework and system for algorithm animation," *IEEE Computer* Vol. **23**(9) pp. 27-39 (September 1990).

41. Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren, and Clark Verbrugge, "EVolve: an open extensible software visualization framework," *Proc of SoftVis 2003*, (June 2003).

42. Wei Wang, "EVolve: An extensible software visualization framework," *McGill University School of Computer Science*, (2004).