A System for Visualizing and Animating Program Runtime Histories

Zhiqing Liu
Land Software
162 von Neumann Drive
Princeton, N.J. 08540
liuz@cs.nyu.edu

Abstract
Program visualization and animation are well known to be important in helping people understand computer programs. Researchers have investigated a variety of techniques and built many systems to this end. However, the use of program visualization and animation systems is limited in real programming environments due to a number of problems. They, among others, include lack of flexibility and poor user interaction resulted from use of ad hoc techniques. This paper describes a programmable and integrated graphical system for visualizing and animating program runtime histories. It employs a special runtime system to accumulate automatically historical information of program execution and allows users to visualize it through multiple, active views in a post-mortem style. Program animation is then integrated into these views by adding a time dimension. Underneath is an animation description language using which we can specify all these views systematically. This paper describes the key features of the system, its primary design considerations and its implementation techniques.

1. Introduction
Program visualization and animation are well known to be important in helping people understand computer programs. (We use the terms visualization and animation to refer to graphical techniques for viewing static and dynamic program entities respectively.) Many systems (e.g., [2][4][5][7][9][12][15][16][17][19][24][25][26]) have been built to this end. However, use of such systems is still confined to classrooms and research labs, and rare in real programming environments. This can be attributed to a number of factors, which, among others, include lack of flexibility and poor user interactions resulted from use of ad hoc techniques.

We can observe these problems from a number of perspectives. Users of current systems usually require to annotate their programs explicitly with either print statements or similar routines to retain interesting events. Techniques are often inadequate in specifying program entities for visualization and animation. When an animation is running “live” with program execution, users have little control of its speed, direction or granularity. And the whole process of program annotation, execution, and animation must be repeated to alter an animation view, which makes any refinements painful.

To address these issues in a systematical way, we have designed and implemented a programmable and integrated visualization and animation prototype for SETL programs, to which we shall refer as SVAS (SETL Visualization and Animation System). This system visualizes and animates programs in a post-mortem style, after retaining runtime histories by using a special-purpose runtime system (persistent runtime system). It then provides a number of active views to help users visualize the runtime histories at an arbitrary program execution moment, both in textual and graphical forms. Additional views (as well as those provided by the system) can be programmed systematically by using a powerful description language. And finally, SVAS achieves program animation on these views easily by adding a dimension of execution time.

We organize the rest of the paper as the following: Section 2 discusses a persistent runtime used in our system and its capability for retaining program runtime histories. Section 3 describes a few program visualization views as well as their interactions with users. Section 4 extends these static views into program animation and discusses its controls in terms of speed, direction and granularity. Section 5 covers an animation description language for specifying program entities upon which our visualization and animation views are built. Section 6 details other primary design and implementation issues of the system. We conclude this paper with a comparison of related work in Section 7 and a brief conclusion in Section 8.
2. Persistent runtime system

It is necessary to capture its runtime information to visualize and animate a program. The way that this is done is crucial to a program visualization and animation system, partly because it can be the most challenging part of the whole system, and partly because it affects the system’s flexibility, interaction style, and ultimately its usability. To maximize the usability of our system, we have chosen to defer building up graphical views for program entities until runtime histories are available. Program visualization and animation are thus performed in a post-mortem, whose advantages will become apparent in the next few sections.

SVAS is built upon a persistent runtime system (PRS), which supports efficient capture of and fast access to fine-grained, multiple versions of runtime state along with normal program execution[14]. Programs running on PRS do not need annotations. Their executions are no different from those running on an ordinary runtime to end users, except that multiple versions of runtime state are retained and can be accessed later, and that possibly more time and memory resources are used. The runtime information retained for a particular programming language depends on its runtime execution model. For an imperative language, this includes updates in both program variables and the execution stack, as well as execution traces. PRS usually retains runtime information at the granularity of program source lines, while it also supports both finer-grained (e.g., at the semantic operation level) and coarser-grained (e.g., at the procedure level) granularities.

The implementation of PRS employs a recent technique of making data structures persistent done by Driscoll, Sarnak, Sleator and Tarjan[8], to which is referred to as the node-splitting method. It splits each node of original data structures into a series of persistent nodes, each carrying some extra fields and a timestamp. Any two consecutive timestamps specify a timeslice in which a persistent node is valid. These persistent nodes are organized in such a way that any node in persistent data structures points only to those nodes with subset timeslices. This property allows access operations in persistent data structures to perform similarly as in ordinary data structures and thus have same time and space complexities. Starting with an access, an update operation does not modify data in place. Instead it stores the new value into one of the extra fields of the persistent node if available, or splits the node into a new node and updates the predecessors of the node, which may cause cascading splits and updates. By restricting the number of predecessors and providing enough number of extra fields matching the number of predecessors, the node-splitting method accures an amortized cost of no more than $O(1)$ both in time and space for each access and update operation.

The core technique of PRS implementation is to use persistent data structures to implement its memory component, by eliminating conventional procedure activation stacks and allocating runtime objects from heaps completely. We organize these runtime objects using persistent data structures and perform persistent updates in the place where overwrites are ordinarily required, following the node-splitting method. There is a top-level array serving as starting reference points to valid information for a runtime history. The $i$th position of this array stores the stack pointer at the $i$th timeslice so that runtime operations can be initiated in $O(1)$ time.

PRS uses a global clock to represent the advance of program execution and signify the entering of new versions of runtime state. We thus can control the recording granularity of PRS simply by manipulating the clock speed. This clock also serves as a synchronization barrier of runtime history recording so that information retained for each timeslice is a complete snapshot of runtime state to which execution can always backtrack. This allows us to run a program with a coarse-grained recording granularity first and refine interesting parts later when necessary, which makes our system more practical for long-run programs.

PRS is efficient in retaining program runtime histories. It has only a linear slowdown compared to ordinary runtime systems and its implementation for the SETL programming language[23] is about half speed of a conventional implementation. It is also efficient (in $O(1)$ time) to access historical information at an arbitrary timeslice or resume program execution from that timeslice. Memory usage of PRS can be substantial but manageable by using different recording granularities. Since PRS supports SETL currently, so does our program visualization and animation system, which will be described in details below.

3. Visualizing program runtime histories

3.1 Multiple views

It is difficult to visualize a program runtime history in one view due to the amount of information involved. SVAS provides multiple views to show different portions of interests or from different perspectives at the same time.

Two essential pieces of information for visualizing a program are its source code and execution trace. They show a static view of the program and its top-level dynamic behaviors respectively, as illustrated in a main window of our system in Figure 1. On the left is a conventional source code view. On the right is an execution trace view whose horizontal axis represents program execution time that is divided into many timeslices. Both of these views share a vertical axis representing lines of program source code. We have designed the execution trace view to show complete information of an execution trace: A rectangle mark (trace-
mark placed at location \((x, y)\) represents that the \(y\)th line of the source code was executed in the \(x\)th timeslice. Because of screen limitations, the trace shown in this view can not be complete and detailed at the same time, unless in the most trivial cases, and usually represents a uniformly distributed sample.

Our system also provides a textual variable view for visualizing program variable values at a selected timeslice. (The selected timeslice is special to our system and will be referred to as the current timeslice, abbreviated as CT, whose use in animation will be discussed in the next section). Figure 2 shows a partial snapshot of this view in which program variables are displayed with their values that were valid at CT. Graphical views of program variables (as well as other program runtime entities) are also available in our system by mapping textual values into graphical primitives such as points, lines and so on, as in many other systems. Because of the implicit and possibly complicated semantics behind and therefore the lack of universal mappings, we do not intend to provide a complete graphical primitive library, and therefore graphical views are usually customarily built and added to the system by users (see Section 5 for additional details). As an example, our system maps variable data (a tuple of integers as shown in Figure 2) into a set of points where the coordinates of a point consist of an integer item and its index in the tuple. Figure 3 shows a snapshot of this.

Also shown in this figure is a slightly more complicated picture representing the same graphical mapping of SETL expression data \(\times 3\). This demonstrates another important feature of our system to evaluate arbitrary expressions and visualize the evaluation results. Expressions in visualization are evaluated by PRS, based on program runtime state.
at CT. This allows us to visualize individual program entities (e.g., variables) as well as complex expressions of and relations among them. More details will be discussed in Section 5.

3.2 Active views

Graphical views in our system are active, i.e., able to accept and respond to users’ manipulations. In a trivial case, users can adjust the source code view by scrolling vertically and horizontally. They can also adjust the execution trace view by dragging the three scrollbars located below the view. The second and the third scrollbars together specify a portion of the full execution trace shown in the view. Initially, they points to the first and the last timeslice respectively selecting the whole trace (see Figure 1). Dragging the two bars closer effects a zoom along the temporal dimension, i.e., selecting and displaying a smaller portion in more details, as shown in Figure 4. The first scrollbar (CT bar) specifies CT. Dragging CT bar or clicking a timeslice in the execution trace view will change CT.

Users of our system can also manipulate the way in which variable values are printed by using three printing modes, namely print, display and zoom. Print shows the value of a variable as ordinarily printed in programs, while display shows the same value as a tree-like structure with proper indents and thus produces an attractive print for compound values. Unlike print and display which both treat a value as a single object, zoom treats it as a tree of objects, in which simple values are leaf nodes and compound structures are internal ones, as illustrated in Figure 5. When zoom is applied to an internal node of a value, the sub-value trees rooted by the node is elided. For example, an elided set is displayed as “{...}”. A further printing on an elided value is applied to each of its sub-values, which allows users to visualize complex variables interactively and incrementally. These printing modes are always reversible. A simple illustrating example of these printing modes is shown in Figure 6.

4. Animating program runtime histories

As we have seen in the previous section, SVAS supports multiple views for visualizing program runtime histories. A crucial property of these views is that they are coordinated by sharing a same execution moment, as represented by CT. When an update on CT occurs, (e.g., by dragging the CT bar below the execution trace view) all these views are updated simultaneously and automatically to reflect the change. When changing CT smoothly and continually, this coordination naturally introduces program animation.

For example, the source code view highlights in reverse video the source line that was executed at CT, changing CT would result in animation of source line execution in the context of source program. Similar animation occurs in the

```
zoom:    print:
  {...}   {{[a b] 1}  {{[a c] 2}  {{[b c] 3}}
zoom+zoom: zoom+print
{{[a b] 1}  {[a c] 2}  {[b c] 3}}
zoom+zoom+zoom:
display: zoom+zoom+zoom:
{{[a b] 1}  {[a c] 2}  {[b c] 3}}
```

Figure 4. Zoom effect on an execution trace

Figure 5. Variable values are organized as trees of objects in printing

Figure 6. Printings under different modes for value {{[a b] 1} {[a c] 2} {[b c] 3}}
execution trace view with tracemarks highlighted in a different context. Animation also occurs in the textual variable view reflecting variable value updates and any graphical views whose graphics properties depend on dynamic program entities.

Users’ manipulations on a view affect its visualization as well as animation. As a simple case, zooming a portion of an execution trace limits the animation on the zoomed portion. More importantly, selected textual printing modes for visualizing a variable determine its animating patterns. Therefore, simple combinations of our printing modes offer conveniently a large number of possibilities to present variable animation.

Use of a shared timeslice to bridge program visualization and animation provides additional advantages for the control of animation, because controlling animation is now equivalent to controlling the change of CT, in terms of direction, speed and granularity. Animation can be performed either forward or backward depending on the direction in which CT changes. Specifying the step between two consecutive animation frames, animation granularity can also be controlled in our system. The finest-grained control comes from dragging the CT bar step by step and CT will change to the next or previous timeslice depending on the dragging direction. Coarser-grained controls are available in both the source code view and the textual variable view. Clicking a source line changes CT to the next or previous timeslice on which this source line was executed, depending on the mouse buttons used. Similarly, clicking a variable name changes CT to a timeslice at which this variable had a different value.

5. An animation description language

While the graphical views described in the last two sections may seem ad hoc, they are built systematically by using an animation description language which we have developed to describe program entities for animation purposes. We shall refer to this language as SADL (SETL Animation Description Language). Due to the scope of this paper, we can not describe the language in complete details. Instead this section discusses briefly its purposes and features, and explains how the language can be used to specify those views described above.

While many techniques have been developed in program animation to specify mappings from program entities to graphical primitives, few are available to allow users to specify those program entities systematically. SADL is a description language to address this. It is designed to describe program runtime entities (variable values, information of the execution stack, their histories, their relations, and so on) in a systematic way such that an animation can be specified easily. Since our system animates SETL programs, SADL naturally shares the syntax and semantics of SETL.

SADL describes an animation view by using a tuple value of two parts: [StaticExpression, DynamicExpression]. Specifying the static part of the view, the static expression is evaluated once when the view is first registered in SVAS. The dynamic expression specifies the dynamic behavior of the view and gets evaluated whenever CT changes. Evaluation results are returned in string format to graphical clients associated with the view. SADL expressions can be ordinary program variables, its special variables (all with a distinguishing $ prefix), and expressions of them. Some examples of the special variables are $SourceCode denoting the source code of the program for animation, $ExecutionTrace denoting its execution trace history (represented as a tuple of source line numbers), and $Timeslice denoting CT. (A complete listing of the special variables and their definitions are given as an appendix at the end of the paper.) We put each of these variable into one of two categories: static and dynamic, based on whether its value depends on $Timeslice or not. Variables depending on $Timeslice are dynamic and the rest are static. For example, $ProcedureActivationDepth denotes the depth of procedure activations at CT and is thus dynamic; while $ExecutionTrace is independent of any particular timeslice and therefore static.

Given the SADL constructs described above, it is now simple to describe our animation views discussed in the previous two sections. The source code view has the description [$SourceCode, $Timeslice]. The execution trace view has the description [$ExecutionTrace, $Timeslice]. The textual variable view has the description [OM, $ActiveVariables], where OM is a special SETL symbol designated to undefined values. The two graphical views shown in Figure 3 have the descriptions [OM, data] and [OM, data*3] respectively. More complicated views can be described similarly.

6. Design and implementation issues

Our system was originally designed and implemented as a visual program debugging environment[13]. Because the fact that it relies entirely on visualization and animation techniques to help users understand programs, verify or refute debugging hypotheses, and identify bugs, SVAS fits well as an program visualization and animation system.

Since it is not practical for a system to provide complete and perfect views for animating arbitrary program entities based on our understanding, we have designed SVAS as an open and programmable system. While providing a small number of predefined views for most common-used cases, our system leaves out many graphical views. It instead allows users to build their own views through using SADL. This is reflected in the overall structure of the system,
by using simple but versatile printing.

When a view is added into the system (system predefined views can be regarded as added at the initialization time), a two-way communication channel (e.g., pipes and sockets) is set up between the view and the animation control stub. The view then passes its animation description to the stub, which evaluates the static part and returns the result immediately. The stub will keep the dynamic expression, evaluate it and return the result to the view whenever CT changes. On one hand, the view is responsible to parse the values returned and map them to graphics properly. On the other hand, it may initiate an update on CT or inform the animation stub to quit, which will then terminate its communication channel and release its dynamic expression stored.

Since large amounts of runtime information can be accumulated in our persistent runtime system[13][14], the design of predefined views are critical to the usability of the system. One of our primary design decisions here is to provide natural and strong orientations so that users can freely explore the animation. Following the conceptual structures of a runtime history, we organize it as a three-dimension information space, whose three dimensions are time, stack and variable. At the top level, execution time divides a runtime history into timeslices. Each timeslice has a number of stack frames, which are then divided into variable values. We provide a global view of runtime information in the main window, which also serves as a reference point for detailed animation. Since variable animation is often desirable, we provide it in the textual variable view by using simple but versatile printing.

7. Related work

As a system for program visualization and animation, SVAS shares important features of and is influenced by many earlier systems. (Excellent reviews of these systems can be found in [18], [20] and [22].) It employs a number of new techniques that make our system unique and interesting.

Figure 7. System Structures of SVAS shown in Figure 7. The kernel of SVAS is PRS described in Section 2. On its top is a stub for animation control. When a view is added into the system (system predefined views can be regarded as added at the initialization time), a two-way communication channel (e.g., pipes and sockets) is set up between the view and the animation control stub. The view then passes its animation description to the stub, which evaluates the static part and returns the result immediately. The stub will keep the dynamic expression, evaluate it and return the result to the view whenever CT changes. On one hand, the view is responsible to parse the values returned and map them to graphics properly. On the other hand, it may initiate an update on CT or inform the animation stub to quit, which will then terminate its communication channel and release its dynamic expression stored.

Since large amounts of runtime information can be accumulated in our persistent runtime system[13][14], the design of predefined views are critical to the usability of the system. One of our primary design decisions here is to provide natural and strong orientations so that users can freely explore the animation. Following the conceptual structures of a runtime history, we organize it as a three-dimension information space, whose three dimensions are time, stack and variable. At the top level, execution time divides a runtime history into timeslices. Each timeslice has a number of stack frames, which are then divided into variable values. We provide a global view of runtime information in the main window, which also serves as a reference point for detailed animation. Since variable animation is often desirable, we provide it in the textual variable view by using simple but versatile printing.

8. Conclusion

Using a timeslice to bridge visualization and animation and using a powerful language to describe it are two key techniques of our system. The use of a shared timeslice has many advantages. It simplifies animation and integrates it into visualization in one single view, which reinforce the concept that the dynamic behaviors of a program are merely its static behaviors changing over time. The use of a shared timeslice also makes full animation control possi-
ble. These advantages are further enhanced by the use of SADL, which not only helps users describe animation succinctly and formally, but also makes SVAS an open system that is easy to extend.

Our informal experience with SVAS shows that it provides simple and effective program visualization with flexible and detailed animation control. The use of SADL encourages a more systematic way to describe and refine animation views. All these techniques result from the post-mortem animation style of our system, which is made possible by the use the persistent runtime system.

Acknowledgments

The author would like to thank Jack Schwartz, David Bacon, Bob Paige, Alan Siegel, Ed Schonberg, Robert Dewar and Ken Perlin for their invaluable comments on the work.

References

Appendix. Special Variable Definitions of SADL

$CompleteHistory := [$ProgramName, $SourceCode, $History] static
$ProgramName := string static
$SourceCode := [$SourceLine*] static/dynamic
$SourceLine := string static/dynamic
$LineNumber := integer static/dynamic
$History := [$HistorySlice*] static/dynamic
$HistorySlice := [#LineNumber, $ProcedureActivation] dynamic
$TimeSlice := integer dynamic
$ProcedureActivation := [$ProcedureActivationRecord*] dynamic
$ProcedureActivationRecord := [$ProcedureName, $Variables] dynamic
$ActiveProcedureActivationRecord := $ProcedureActivationRecord dynamic
$ProcedureActivationLevel := integer dynamic
$ProcedureActivationDepth := integer dynamic
$ProcedureName := string dynamic
$Variables := {[$VariableName, $Variable]*} dynamic
$ActiveVariables := $ActiveProcedureActivationRecord (2) dynamic
$VariableName := string dynamic
$Variable := [$VariableType, $VariableValue] dynamic
$VariableType := string dynamic
$VariableValue := string dynamic
$ExecutionTrace := $TraceOfLineNumber static
$TraceOfLineNumber := [s(1) : s in $History] static
$TraceOfSourceLine := [$SourceCode (l) : l in $Trace] static
$HistoryOfProcedureActivation := [s(2) : s in $History] static
$HistoryOfProcedureActivationDepth := [#a : a in $HistoryOfProcedureActivation] static
$HistoryOfActiveProcedureActivationRecord := [a(#a)(2) : a in $HistoryOfProcedureActivation] static