Visualizing the Behavior of Dynamically Modifiable Code *

Bradley Dux Anand Iyer Saumya Debray David Forrester Stephen Kobourov
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
Email: {dux, anand, debray, forrestd, kobourov}@cs.arizona.edu

Abstract

Recent years have seen an increased recognition of some of the advantages offered by dynamically modifiable code, i.e., code that changes during the execution of the program. In its full generality, it can be very difficult to understand the behavior of such self-modifiable code. This paper describes a system that graphically displays the execution behavior of dynamic code, focusing on code modifications and their effect on the structure of the program, i.e., the call graph and control flow graphs of functions. This can help users visualize the structure of runtime code modifications and understand the behavior of dynamically modifiable programs.

1 Introduction

Dynamically modifiable software refers to software whose executable code changes as the program executes. These changes may involve a change in the set of memory addresses containing executable instructions (i.e., when new code is generated during the program's execution) and/or a change in the contents of one or more locations within the program's executable code space (i.e., when an existing instruction is modified to a different instruction). Sometimes referred to as "self-modifying code," dynamically modifiable programs have long been deprecated for being difficult to write, understand, and maintain. In recent years, however, there has been an increased awareness of the advantages dynamically modifiable software can offer under certain situations. Applications of runtime code generation and modification include just-in-time compilation [1], dynamic code specialization and optimization [2, 8, 12], dynamic decompression of compressed code [4], and security enhancement [7, 16]. Indeed, several authors have proposed programming language extensions to facilitate the writing of dynamically modifiable code [5, 13], while others have looked at extending tools to cope with runtime code modifications [9].

A significant problem with dynamically modified programs is that their behavior can be complex and difficult to understand. The reason for this is that our conventional models of software structure are static. The behavior of a program is understood in terms of the behaviors of the functions comprising the program and their interactions; the behavior of each function is understood in terms of its constituent instruction sequence, which is generally taken to be fixed. Such static models break down when code can change at runtime. This introduces an additional dimension of complexity into the program analysis process: for example, program analyses dealing with a function call to some address can no longer simply use a precomputed summary of the behavior of the function at that address.

The difficulty of understanding, debugging, and maintaining dynamically modifiable software suggests the need for tools to help users understand the behavior of such code. This paper describes a first step we have taken in this direction. Our tool uses a replay mechanism to graphically display the effects of runtime code modification on the program's call graph as well as the control flow graphs of individual functions.

The remainder of this paper is organized as follows. Section 2 provides a simple model for dynamically modifiable code that forms the basis for our visualization tool. Section 3 describes salient aspects of this tool. Section 4 describes related work, and Section 5 concludes.

2 Modelling Dynamically Modifiable Code

This section presents a simple conceptual model for dynamically modifiable code.

There are two different kinds of entities we are concerned with: *memory locations* and *functions*. Memory

*This work was supported by the National Science Foundation under grants CCR-0073394, EIA-0080123, and CCR-0113633.

locations are the entities that are actually modified during program execution, while functions are the entities that programmers base their understanding of programs on. Given a function f in a program, let $locs(f)$ denote the set of memory locations occupied by function f. Dynamic code modification of a program changes the contents of one or more code locations during execution. In general, when the contents of a code location changes, the instruction at that location changes, and therefore has a different runtime behavior. This means that the function containing that location also has a different runtime behavior. Semantically, therefore, the function associated with that memory location is now a (mathematically) different function than what it was before the change. To understand dynamically changing code, we have to map runtime changes to memory locations to corresponding changes to functions. To this end, we define a relation $\sim$ between pairs of functions that captures the intuition of "sharing a memory location:"

$$f \sim g \quad \text{iff} \quad locs(f) \cap locs(g) \neq \emptyset.$$

The relation $\sim$ is reflexive and symmetric, but need not be transitive. Let $\sim^\star$ denote the transitive closure of $\sim$: this is an equivalence relation, and partitions the functions in a program into clusters. These clusters have the property that the functions in each cluster "directly or indirectly share locations," in the sense that f and g are in the same cluster, i.e., $f {\sim^\star} g$, if and only if there is a chain $f_1, \ldots, f_n$ of location-sharing functions:

$$f \sim f_1 \wedge f_1 \sim f_2 \wedge \cdots \wedge f_n \sim g.$$

This means that at runtime, whenever there is a change to a program's code for some function f, the new function g that results from this change must satisfy $f {\sim^\star} g$, i.e., be in the same cluster as f. Such clusters of location-sharing functions form the basis for our approach to visualization of dynamically modified code, described in the next section.

3 Visualizing Dynamically Modified Code

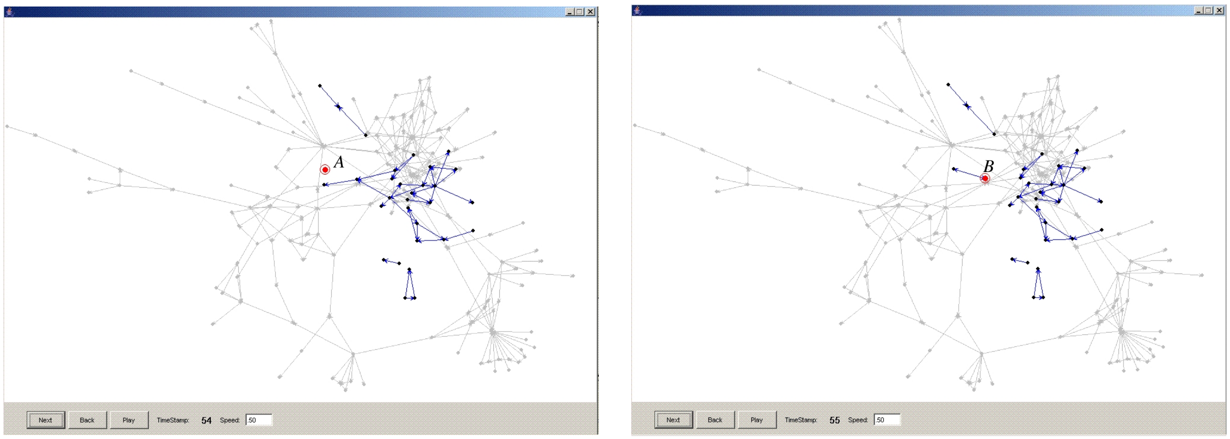
We have built a prototype tool, based on the ideas described in the previous section, for visualizing dynamic software modifications. Our tool, written in Java, is based on the `graphael` system[6], with some modifications to deal with the scale issues we encountered when dealing with self-modifying programs.

The motivation for this tool arose from some work being done within our group on using runtime code mutation for code obfuscation purposes [10]; however, the ideas are general, and are applicable to other dynamically modifiable programs as well. The visualization is organized in terms of *time-slices*, where each time-slice corresponds to a set of related changes to the code of a single function (and may not necessarily correlate directly with elapsed runtime). The visualizer uses a log file of code modifications to organize the initial organization of the program's graphical representation, and to then display the effects of code modifications.

The GUI. Our program visualization tool currently provides a simple graphical user interface. The visualization is controlled using three buttons, *play/pause*, *next*, and *back*, together with a *speed* parameter, all of which appear below the main window. The *play* button displays the changing graph step by step, like an animation, with the time-slice number being displayed on the bottom. When the animation is playing, the *play* button becomes the *pause* button, which enables the user to stop the display at any time. When paused, the *pause* button becomes the *play* button. The *next* and *back* buttons enable the user to go forward or back, respectively, by a single time-slice, every time the button is pressed. After moving backward or forward as desired using these buttons. the user can resume the animation by pressing the *play* button. The *speed* value determines how fast the animation is played i.e time for which a single time-slice is displayed. The user can right-click on a cluster node to view the control flow graph of the function in that cluster. Double clicking on a given node displays the name of the function currently in that cluster.

Cluster Call Graphs. As mentioned earlier, our tool uses a fixed precomputed vertex layout for all the vertices that will be encountered during the evolution of the program. The initial cluster call graph for the program is then drawn based on this layout. Note that since the vertex layout may contain some vertices that do not come into existence until after some number of code mutations, the initial cluster call graph may not include all of the vertices used in the initial layout determination. The vertex layout remains fixed through the duration of the program's execution, and changes to the cluster call graph are reflected in changes to the edges in the graph displayed.

In general, not all functions in a program will be subjected to code modifications. Since we focus primarily on visualizing the effect of dynamic code changes to the structure of the program, we want to relegate such static functions to the background as much as possible (they cannot and should not be eliminated entirely, since they can and do influence the behavior of dynamic code). To this end, we classify the edges in the cluster call graph into two kinds: "cluster edges," i.e., edges where at least one endpoint is a cluster that is modified at runtime;



(a) The call graph before a code modification step; (b) The call graph after a code modification step.

Figure 1. Visualizing the effects of a code modification step on the call graph of a program. Cluster edges are dark/blue, non-cluster edges are light/gray. The labels 'A', 'B' do not occur in the actual image, but have been added for expository purposes.

and "non-cluster edges," where each endpoint is a static function that is never modified during program execution. Our tool displays non-cluster edges in light gray, while cluster edges are displayed in dark blue. This lets cluster edges stand out conspicuously while non-cluster edges are visible, but are not obtrusive.

Figure 1 shows the cluster call graph for a program, as drawn by our visualization tool, at the beginning and end of a single time-slice, i.e., before and after a single code modification. The cluster that is "active" at the beginning of the time-slice, i.e., about to undergo a code modification, is labelled 'A' in Figure 1(a), while the cluster that is active at the end of this time-slice is labelled 'B' in Figure1(b). Notice that the code modification to cluster A during this time-slice changes the call graph edges. At the beginning of the time-slice, the function in cluster A does not call any other function, nor is it called by other functions. Hence A is not connected by any edges to any other vertices. In the next step the function in cluster B changes. The three edges that connected cluster B in the previous time-slice disappear, and a new edge connect it to cluster A. This is representative of the changes that occur to the call graph as the functions in the clusters are modified at runtime.

Not all runtime code modifications result in changes to the (cluster) call graph of the program. Our tool uses the color of a vertex to indicate whether or not the corresponding cluster is having its code modified at any given time-slice:[1] The vertex that is being modified at a given time-slice is shown as a large red circle. This serves to indicate where the actual changes are occurring. The other cluster vertices are shown as medium-sized black circles.

While experimenting with our tool, we found that sometimes, specific sequences of code modifications would repeat over and over again. For example, a function f might be edited to a function g, which might be edited to another function h, which might then be edited back to f to start the edit sequence over again. When a group of N contiguous repetitions of a sequence of code modifications w is found, instead of repeatedly displaying the effects of the sequence w on the program, we display the effect of w once, together with a tag indicating that this effect is repeated N times. Identifying such repeated edit sequences and tagging them as such in the image of the call graph displayed by the tool turns out to be very useful, both from the perspective of understanding the behavior of the program, and also from an efficiency perspective.

Control Flow Graphs. The cluster call graph discussed above gives the user a high-level view, in terms of procedure clusters, of where code edits occur as the program executes. This view can then be refined, as desired by the user, by clicking on a vertex in the cluster call graph. This results in the call graph for the currently executing function within that cluster to be displayed, as shown in Figure 2.

[1] We assume that exactly one vertex is being modified at any given time-slice. Concurrent independent modifications to multiple vertices can be handled by serializing them arbitrarily.

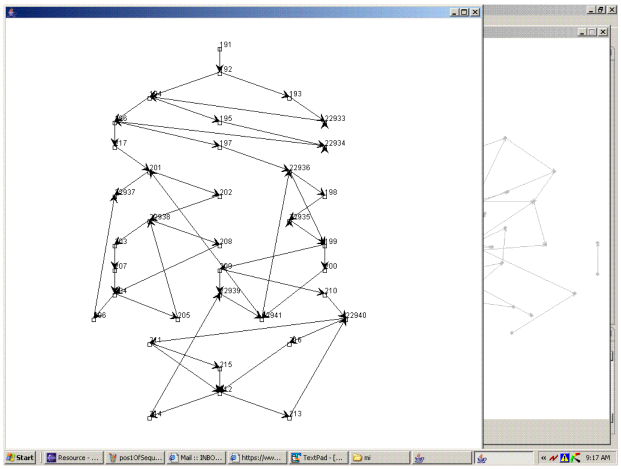


Figure 2. Displaying the control flow graph of a function.

4 Related Work

We are not aware of any other work on visualizing the effects of dynamic code modification on code structure. There has been a significant amount of research, in recent years, on code that is created or modified at runtime [1, 2, 4, 7, 8, 12], and a number of researchers have proposed programming language extensions to facilitate the writing of dynamically modifiable code [5, 13] or manipulate such code [9]. However, none of this work addresses the issues of visualizing or understanding the effects of runtime modifications. to the code of a program. There is a large body of work on software visualization tools [3, 11, 14, 15], but none are concerned with self-modifying code.

5 Conclusions

Recent years have seen growing interest in software system where the code that is executed is not a fixed, static body of machine instructions, but can be changed at runtime, e.g., by generating new instructions or modifying existing instructions. Understanding the behavior of programs where the code changes during execution can be quite nontrivial. This paper describes a prototype tool that allows users to visualize the effects of runtime code changes on the structure of the program.

References

[1] A.-R. Adl-Tabatabai *et al.* Fast, effective code generation in a just-in-time Java compiler. In *Proc. PLDI-98*, pages 280–290, June 1998.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proc. PLDI-2000*, pages 1–12, June 2000.

[3] C. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proc. SoftVis-03*, pages 77–86, 2003.

[4] S. K. Debray and W. Evans. Profile-guided code compression. In *Proc. PLDI-02*, pages 95–105, June 2002.

[5] D. R. Engler and T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Proc. ASPLOS-VI*, pages 263–271, 1994.

[6] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. GraphAEL: Graph animations with evolving layouts. In *11th Symposium on Graph Drawing*, pages 98–110, 2003.

[7] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting self-modification mechanism for program protection. In *Proc. COMPSAC-2003*, pages 170–181, Nov. 2003.

[8] M. Leone and P. Lee. A declarative approach to runtime code generation. In *Proc. Workshop on Compiler Support for System Software (WCSSS)*, Feb. 1996.

[9] J. Maebe and K. De Bosschere. Instrumenting self-modifying code. In *Proc. AADEBUG-2003*, pages 103–113, Sept. 2003.

[10] P. Moseley and S. K. Debray. Software protection via dynamic code mutation. Technical report, Dept. of Computer Science, University of Arizona, 2004.

[11] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, Mar. 1990.

[12] F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proc. ICCL-98*, pages 132–142, 1998.

[13] M. Poletto *et al.* 'C and tcc: A language and compiler for dynamic code generation. *ACM TOPLAS*, 21(2):324–369, Nov. 1999.

[14] B. A. Price, I. S. Small, and R. M. Baecker. A taxonomy of software visualization. In *Proc. 25th Hawaii Int. Conf. System Sciences*, 1992.

[15] G.-C. Roman and K. C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12):11–24, 1993.

[16] D. D. Zovi. Security applications of dynamic binary translation, Dec. 2002. Bachelor of Science Thesis, Dept. of Computer Science, University of New Mexico.