Challenges in building live programming environments for general-purpose programming

Suman Karumuri, suman.karumuri@gmail.com

Live programming aims to improve program comprehension by linking a dynamic semantic representation of a program with the static textual code. In this paper, we draw on past research in program comprehension/software visualization to identify the key challenges and possible solutions to build live programming environments that are suitable for general-purpose programming.

I. Introduction

Today's software is complex. It involves multiple threads or processes, complex locking behavior, non-determinism and large code bases implemented by teams of programmers. It is often reactive, responding to external or user events in an asynchronous manner. While understanding static structure of the program is a daunting task, understanding the dynamic behavior of complex interacting system is often worse. Developers are always on a constant look out for tools to bridge these gulfs of execution and evaluation [3].

Live programming environments strive to bridge these gulfs by letting the user specify the program in a concise textual form, while the environment provides an interactive semantic representation of the program, as the programmer is typing the program. For the purposes of program comprehension, the 4 liveness levels proposed by Tanimoto [5] translate into the following features in a live programming environment respectively (a) Live evaluation of the code as it is being written. (b) A semantic visualization of the code. (c) Incremental semantic visualization of code for incremental edits. (d) Interacting with the semantic visualization to update the code. Bret Victor has rekindled interest in live programming environments by showing several interesting applications of a Level 4 live programming system [6]. Most live programming environments often use the output of the program, as its semantic visualization.

Researchers in the field of program comprehension and software visualization have worked on similar problems for large general-purpose systems. They have observed that programmers working on complex software systems often use several high-level abstractions to think and answer questions about various aspects of their systems. Sometimes these high level abstractions may be incomplete or incorrect. Over the years the program comprehension community has used these insights to identify the challenges and has developed tools and techniques to answer different questions about a program's behavior.

In the next section, we will identify some key challenges in extending these liveness features to a general purpose-programming environment by drawing from the past work of the program comprehension and software visualization community.

II. Key Challenges

A. Live evaluation of code

Live evaluation of code is useful and is widely used in day-to-day experimentation. For example, it is widely used in console evaluation during a debugging session (to understand the run time structures, to explore an unknown API or for one off computation tasks), by web designers to tweak css values in Firebug, Web Inspector etc. While live evaluation is widely used, it is still a small part of overall software development life cycle and day-to-day software development. Can we extend the benefits of live evaluation to other software development tasks like software maintenance and refactoring?

The other challenge in live evaluation of code is related to (development and setup) cost. For example, a programming language (like C) may not be amenable to live evaluation or the tools may lack support for live evaluation (ex: the build system being too complex, takes a long time, a lot of resources, or resource limitations in an embedded system). Even if the language and tools support live evaluation, the code may be setup in a way that makes it tedious to give it inputs (ex: tightly coupled interfaces, distributed systems), which may lead to poor adoption of a live programming environment. What attributes of a programming language, library, framework or a tool would make it amenable to live programming? Can technologies like reversible debuggers or automatic code generators (unit tests, mock objects etc.) be applied to improve live programming?

B. Static Semantic feedback

Semantic feedback refers to an intuitive and meaningful representation of the certain aspects of a piece of code, component or system. In some cases, semantic feedback can be simple and straightforward like the result of evaluating a certain expression or the graphical output produced by a certain piece of code. However, if the component is a complex system or a system whose output or function is non-visual the term semantic feedback can mean different things depending on the question being asked [2]. For example, consider a multi threaded web crawler written in Java, where each thread picks a URL from the shared queue, downloads the page from the web and parses the downloaded page to find unvisited URLs to be added the shared queue. In this example, one might want to know the architecture of the crawler, why some URL's on the page are getting dropped, why all the threads in the crawler appear to be stuck or how one can improve the performance/throughput of the crawler. Each of these questions can be answered by an architecture diagram, stepping through the HTML parser in a debugger, thread visualization in DYVISE [7] and using network traces from X-TRACE [4][8]

respectively. In each case, different types of semantic feedback are required to answer different questions about the system. What does semantic feedback mean in different contexts?

Steve Reiss [2] have proposed several systems that build models of programs and provide several intuitive visualizations to answer different types of questions about a system or a component of a system. Similarly, can we build models of programs that can provide semantic feedback to answer specific questions about a broad class of systems?

It has also been shown that providing semantic feedback about a system may not be enough to improve program comprehension abilities because, looking is not seeing, programmers need to be trained in using visualization tools because graphical representation is a new language in itself [Petre:1995]. How might we educate users about semantic tools or reduce the barrier to adoption when these tools are difficult to learn? Does standardizing the graphical representation of the semantic feedback help?

C. Incremental semantic feedback

By providing incremental semantic feedback of code, a live programming system would show the semantic differences caused by a set of edits. Tools like Firebug, Web inspector etc. allow users to update a css rule and see it's incremental effect right away. Similarly, GUI builders in Eclipse also show incremental differences after every code change. Unlike high-level visualizations, which have shown to be useful for program comprehension [2][3], the value of incremental semantic feedback for small code changes is unclear since programmers can often maintain the effects of a small edit in their working memory. Is incremental semantic feedback valuable for most code changes? What type of programs would benefit most from incremental semantic feedback?

Incremental semantic feedback is simple to show and easy to understand when (a) there is a one to one correspondence between code and its semantic feedback (a program's output in above examples), (b) the updated code directly effects the semantic feedback and (c) It is easy to compute the semantic difference between the current semantic feedback and the new semantic feedback. In the general case, there may not be a 1 to one correspondence between updated code and the semantic feedback or the new code may have an indirect (*n*th order) effect on the semantic feedback. Further, it may not be easy to compute a semantic difference for a given code change. For example, adding a the thread pool to our sample web crawler will have negligible impact on the architecture of a web crawler but will significantly change its run time characteristics. Is it possible to provide incremental semantic feedback in the general case? For small code changes, can we avoid solving the problem of providing incremental semantic feedback, by augmenting programmers working memory using simple code visualizations provided by tools like code bubbles [1] and code canvas [9]?

D. Updating code by interacting with its semantic feedback

A GUI builder is an example of a program where updating the visualization would update the code. Similarly some features of CAD/CAM programs, photo-editing programs or the macro-recording mode in vim are examples of programs where the code is updated by manipulating its semantic output. However, achieving such tight integration between code and its semantic feedback requires specialized programming in specific end user applications to be used in specific cases.

A lot of tools exist for viewing and updating configuration data in a visual way. Tools like jvisualvm, DYVISE [7] can be used to understand and update the run time characteristics of a Java program like thread state, garbage collector status, CPU and memory load through a visualization. For operational reasons, it is a common practice to build control panels or dashboards that allow monitoring and tuning application specific configuration in a software system. However, no general-purpose tools exist for updating code of program by interacting with its semantic feedback.

What kind of programs would benefit from such tight coupling between code and it's semantic feedback? Why are there no general-purpose tools available that allow a developer to update the code while interacting with its semantic feedback? What layer of the programming stack (programming language designers, compiler/run time implementers, tool builders, library or framework authors) would implement such a tight coupling between a program and its semantic output? Can such tight coupling be implemented in a cost effective manner or be automatically generated?

III. Conclusion

In this paper, we have identified several challenges for improving live programming systems by drawing on work from the program comprehension community. We have identified the challenges in expanding level 1 and level 2 live programming systems to a much wider range of programs. While we have identified some niches where level 3 and level 4 liveness is beneficial, more work is needed to characterize where such features would add value. Since the cost to benefit ratio of implementing the level 3 and 4 liveness seems high at this point, we may need to find cheaper ways to achieve those benefits in the general case. It is exciting to see live programming as the new battle cry for improving program comprehension. I hope that this paper would lead to an exciting discussion on this topic at the conference.

IV. References

[1] Andrew Bragdon et al, Code Bubbles, a working set-based interface for code understanding and maintenance. CHI'10.

[2] Steven Reiss et al. What is my program doing? Program dynamics in programmer's terms. RV'11, pages 245–259, 2012.

[3] Henry Lieberman et al, Bridging the gulf between code and behavior in programming, CHI '95, pages 480–486, 1995.

[4] Rodrigo Fonseca et al, X-Trace: A Pervasive Network Tracing Framework, NSDI'07.

[5] Steven L. Tanimoto. Viva: A visual language for image processing. *J.Vis. Lang. Computing*, June 1990.

[6] Bret Victor. http://worrydream.com/LearnableProgramming.

[7] Steve Reiss. DYVISE: Performance Analysis of production Systems: Research Demonstration, ICSE, 2009.

[8] S Reiss et al. Dynamic detection of event handlers, WODA'08.

[9] R DeLine at al: Code canvas: zooming towards better development environments. ICSE 2010.