A Demonstration of Simultaneous Execution and Editing in a Development Environment

Steven P. Reiss and Qi Xin

Department of Computer Science
Brown University
Providence, RI 02912, USA
{spr,qx5}@cs.brown.edu

***Abstract*—We introduce a tool within the Code Bubbles development environment that allows for continuous execution as the programmer edits. The tool, SEEDE, shows both the intermediate and final results of execution in terms of variables, control flow, output, and graphics. These results are updated as the user edits. The user can explore the execution to find or fix bugs or use the intermediate values to help write appropriate code. A demonstration video is available at https://www.youtube.com/watch?v=GpibSxX3Wlw.**

***Index Terms*—Continuous execution, integrated development environments, debugging, live coding.**

I. Motivation

Programmers often think in concrete terms while writing and debugging more abstract code. They start with an example and write code that handles that example, generalizing the code as they go. They debug with a particular example in mind. The ability to see intermediate results and understand and check code as it is written is central to spreadsheet programming and is used in interactive environments such as MATLAB and in dynamic languages such as Smalltalk or Python. For example, Sharp [28] notes that for Smalltalk, "A useful technique for writing new code is to write most of the code in the Debugger". Bret Victor claims this type of live coding is the preferred way to code [33]. Programmers can do this to a very limited extent with today's traditional Java programming environments using live update, the ability to reload a class and continue execution.

Our prototype tool, SEEDE, provides the ability to see immediately the effect of code changes on execution in a Java environment for real programs and a wide range of different edits, essentially providing continuous execution. It does this within the Code Bubbles programming environment, letting the programmer start a new session at any breakpoint and showing the updated execution as the programmer edits. It also lets the programmer select a test case and view the results of executing it.

A system that does continuous execution while editing needs to meet certain requirements. These include:

- *Performance*. The evaluation needs to be fast enough to be run potentially on each keystroke and to provide feedback within seconds. If feedback is slower, it can either confuse the programmer (by showing older values) or cause unnecessary delays.
- *Non-obtrusive*. Execution feedback should not require substantial work on the part of the programmer. Currently, to use Java live update, programmers need to save error-free code and then either step or continue from wherever the save placed execution (which might be the start of the method or the start of some previous method) up to the point where they were editing. As a first approximation, we wanted to show the new values at about the same point without requiring any extra work by the programmer.
- *Idempotent*. Continuous execution should not actually change any values in the execution or the external environment. Any such changes would make running the code multiple times problematic. For example, a Java method that starts with or contains the code:

  ```
  if (!done.add(input)) return;
  ```

  can only be executed once since the next execution would just return. Java live update does not work in these cases.
- *Error Tolerant*. The intermediate code created by the user will contain both syntactic errors and semantic faults. Note that, especially when creating code, there are likely to be errors later in the method even while the code up to the point of interest (where the user is editing) is correct. Continuous execution must be able to work in such an environment, providing output at least up to the first error.
- *Complete*. The system needs to be able to handle a large fraction of the underlying language. This means being able to handle files, graphics, as well as all the routines that involve native code. It also means handling a wide variety of edits.

Our approach is based on three insights into how to build a practical continuous execution environment. The first is that it is generally sufficient to consider only the execution of a single method within an execution. If SEEDE is used for writing code, then the method to be written should be the focus of attention. If the system is used for debugging, then the user is generally looking at a particular method (for example a unit test case). Concentrating on a particular method rather than the full execution makes the notion of interpreting and saving all values practical.

The second insight is that the execution should be triggered from a breakpoint in an actual run. The environment needed to run a method (i.e. all the associated data structures,

978-1-5386-2684-9/17/$31.00 © 2017 IEEE

current values, etc.) can be large and complex. Associating a SEEDE run with a debugger session lets SEEDE query the debugger to access the current environment, something that could be difficult for the user to manually specify.

The third insight is that the system needs to convey the complete execution including all intermediate results and make it easy for the programmer to navigate within these results. This is primarily for debugging where the programmer will need to follow the execution and understand where and when problems occurred, but is also useful in writing new code.

Our approach operates by combining three interpreters into a single system. This is outlined in Section IV. It includes a viewer as part of Code Bubbles that is automatically updated as edits occur. This can be seen in the example shown in Section III. Limitations of the approach are discussed in Section V. We are currently working on a formal evaluation of the system as described in Section VI.

II. Related Work

The idea of providing immediate execution feedback while coding was central to spreadsheet programming introduced by VisiCalc. The idea was picked up for procedural programming by VisiProg [9], and more recently in the EG extension to Eclipse [5]. These are both illustrated on simple programs and do not scale to real systems nor do they address the much more complex problems posed by real systems with complex data structures, external methods, and concurrency. Victor in his talk on live coding demonstrated a sample framework and challenged the audience to create a real one [33]. A more extreme version of using examples to help coding can be seen in the various programming-by-example systems that have been developed over the years [6,26,31]. The approach is also being used effectively for database interactions using continuous queries [1] and in interactive data exploration tools.

There have been several studies on how programmers debug and on what tools and techniques might be helpful [18,19,34]. These tend to show that the type of information and assistance provided by SEEDE can be helpful.

Since many of the examples cited for continuous execution are effectively test cases, this work is also related to early efforts to integrate testing with code writing as in Tinker [12], and more recent efforts involving continuous testing [27]. The work is also related to incremental execution [13,23] and continuous and incremental program analysis [2,14,22,36].

Java, and hence various Java programming environments, support live update of compiled code [16]. This lets the programmer effectively write code while debugging using the full capabilities of the debugger to examine program state and see the effect of the changes. This facility can be helpful but is also very limited.

One of our goals is to provide an implementation framework that can provide the benefits of live update without the limitations. Java live update requires the programmer to save the edited code (without errors) and then re-execute the current function, possibly stepping through the execution to get to the appropriate point of interest. Our approach tracks the current position in the execution and automatically restores it after an edit. Live update fails completely under many common circumstances including adding or removing a field or method, changing method or field signatures, recursive executions, changes to data structures, changing declared constant values, or compiler errors. Our approach handles all of these, either automatically or with minor user intervention (for example when a new field needs a non-default value for existing objects). Live update is not idempotent, so code in the method that changes the environment cannot be cleanly reexecuted. Again, our approach of executing outside the original environment handles this cleanly. Live update does not handle external I/O, causing multiple occurrences of output and requiring the user to reenter input each time. We address these correctly for the terminal and files. Live update also has problems with synchronization. since doing live update while holding a lock does not release the lock. With our approach locks are only maintained within the simulation, not in the original program.

A number of systems over the years have been capable of showing a full execution and letting the user move backward or forward in time within that execution. EXDAMS was perhaps the earliest example [3]. Early graphical environments such as PECAN let the user step either forward or backward [23]. The algorithm animation system BALSA provided a time slider similar to the one we offer [4]. Among the many more recent debuggers that include similar features are TotalView [7], Elm's time-traveling debugger [17], the Trace-Oriented Debugger [20] and others. Ko's Whyline provided similar capabilities in a question-answering framework [11].

Dynamic updating has been used for maintaining long-running applications. A number of techniques have been developed that take updates and modify the existing system to use the new code [15,30,32]. These require the programmer to identify safe points and concentrate on migrating object implementations. While some of these technologies are useful, most of it is too heavy-weight to be used continually while the programmer is editing. Dynamic object updating has also been at the center of schema updates for object-oriented database systems [29]. Our approach uses appropriate techniques from these system to simulate object migration where necessary.

Sandboxing of files is used extensively for providing security to applications that might be unsafe and served as a motivation and guideline for modeling external events in our framework [8,10,35].

III. Use

The Code Bubbles tutorial program [44] is a simulation of the Romp toy [39]. The tutorial includes several tasks involving fixing the display output, notably to change the color of the magnets and to center the +/- output on the magnet correctly. To use SEEDE on the tutorial example, we start by setting a breakpoint at the start of the drawing routine for the board and then start a debugging run up to that breakpoint.

(The breakpoint can be at any point in the method as long as the method is idempotent up to that point.) Then we right click to bring up the default pop-up menu, and select "Start Continuous Execution".

This creates the uninitialized output display bubble shown in Figure 1a. and starts the continuous execution process. Selecting continuous execution for a test case does this all automatically for that test case, creating a launch, setting a breakpoint, running the code, and then starting SEEDE.

SEEDE monitors changes to all editors in the same working set as the SEEDE bubble. If the user opens new editors, they are monitored as well. Other code is interpreted in its original state from byte code. Continuous execution on edits is triggered automatically by messages from Code Bubbles that indicate edits. The display only shows methods from the set of editors that are available. Other calls, for example, to library routines, are hidden.

The first execution can take significant time since values need to be loaded from the running process and binary files need to be loaded from the class path. Both of these are cached by the SEEDE back end so that subsequent runs are faster. (The actual performance depends on a variety of factors such as the amount of data that is retrieved, the amount of data being passed back to the front end, and the number of instructions executed. In this case, the time is about 5 seconds.) Once the initial execution has completed, the system populates the various components of the continuous execution bubble with the resultant values as shown in Figure 1b.

The default view provided by the evaluation bubble is a tree containing the variables and their values that were computed during the simulated execution. For objects and arrays, the user can expand the tree to see the sub-values. The variables are displayed one frame (method) at a time. The scroll bar at the bottom of the variable display lets the user scroll over the execution by time. Green areas in the scroll bar represent code in the current method; gray areas represent code in called methods. As the user scrolls, the variable values change to reflect their values at that point of the execution. A special variable, *LINE*, shows the current line number at that point. This line is also highlighted in any editor that is open that includes the method. In addition, hovering over a variable in the open editor will show the history of values of that variable up to the current time. This can be seen in Figure 1j.

At the upper right of the window, the system displays the execution status. This can be *PENDING* (waiting for an execution to complete), *RETURN* (routine successfully returned), *COMPILER_ERROR* (execution stopped with a compiler error), *EXCEPTION* (execution stopped with an exception), *ERROR* (a problem in SEEDE, typically a call to a native method), or *TIMEOUT* (execution stopped because it was taking too long, e.g. with an infinite loop).

In addition to the variable window, the SEEDE display can show a graph of the lines executed (Figure 1c) or the call tree or graph (Figure 1d). The call tree can also be displayed as a linear view of the stack over time (Figure 1g). The user can select a value and ask for the data dependence graph for that value. The result is a display of the values leading to it (Figure 1h). In addition, if the program does any file output, the result is shown in an output panel (Figure 1i).

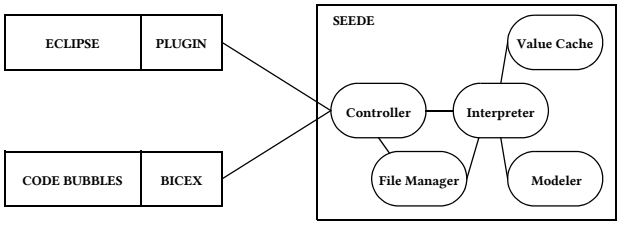


Fig. 2. Overview of the SEEDE architecture.

Experience using SEEDE demonstrated the importance of being able to navigate over time to understand the overall execution, especially when using the tool for debugging. As a result, we have added a variety of navigation options. One can right click on the time scroll bar to go to an inner context. One can go to the next or previous call or the next or previous line. One can click on a value in the call tree or the stack view to see that particular instantiation. One can select a variable and go to the context where it was written. In addition, the call graph, stack, and data dependence views provide navigation options.

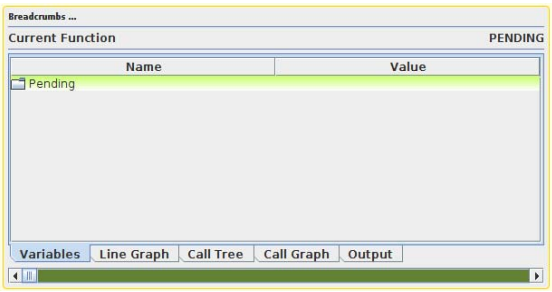
In the example because the breakpoint was set at the start of the paint routine, the continuous execution bubble includes a graphics panel showing the window output as computed by interpreter in Figure 1e. This is generated automatically by SEEDE once it detects that the routine being interpreted is a painting routine. Users can also point to other variables that reflect graphical components and request a drawing window for those as well.

At this point, we can attempt to edit the code and see the result. We first bring up the *drawMagnet* routine in a separate bubble and then change the color setting for *drawOval* from *Color.green* to *Color.red*. In under a second the graphical output view updates to reflect the change. Next we try centering the + or - over the magnet by changing the position passed to *drawString*. After each change, we see the result and can quickly settle the proper change to the coordinates. The final result is shown in Figure 1f.

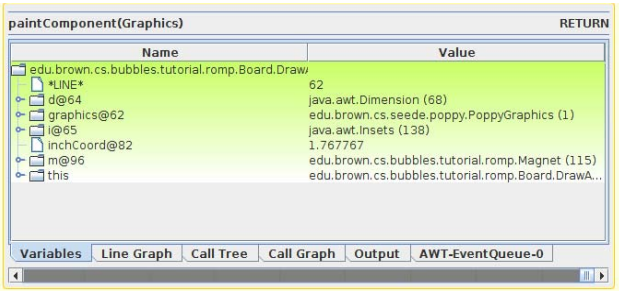
IV. Implementation Overview

SEEDE runs as a separate process, talking to both the front end application and to Code Bubbles' Eclipse-based back end through the Code Bubbles messaging interface [24] as shown in Figure 2, It takes requests from the application and sends execution updates back to it as they become available. It uses the back end to query the value of variables, to understand the Java environment, and to detect changes both to the execution and to files being edited.

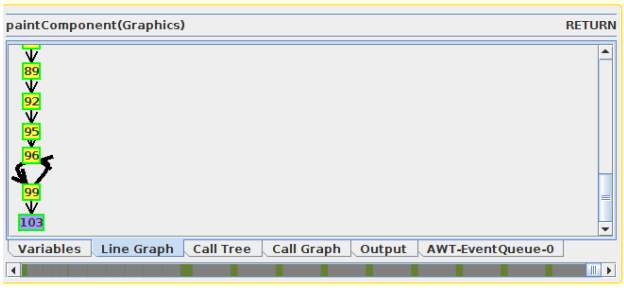
The actual system has several major components. First, there is a value cache. This holds the value of any variable that is accessed by the code that is being run. Moreover, it tracks those values over time, so that it actually stores all the values



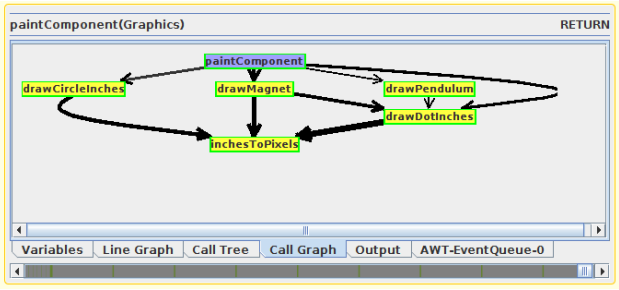
a) Initial view before initialization



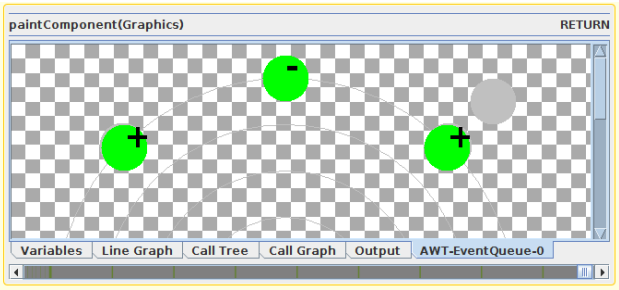
b) Variable view showing time slider and values



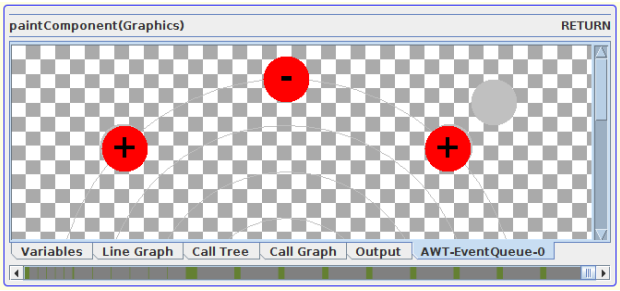
c) Graph of line-number based execution



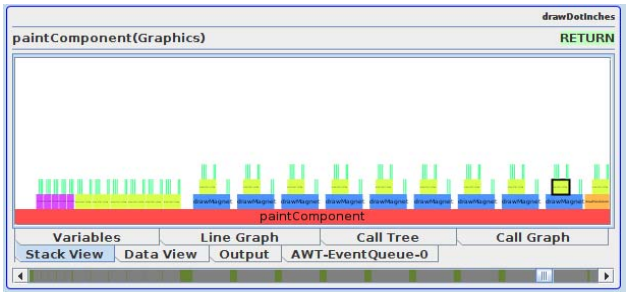
d) Call graph view of execution



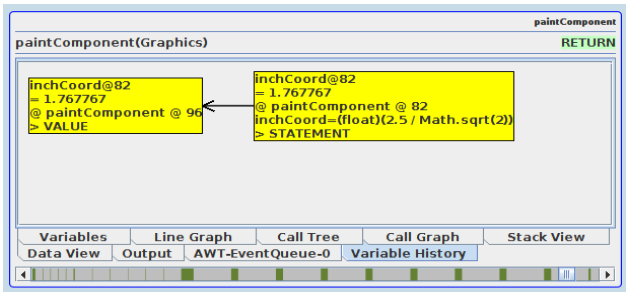
e) Initial graphical output window



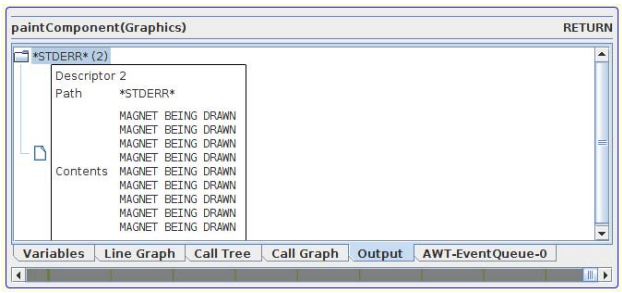
f) Graphical output after editing



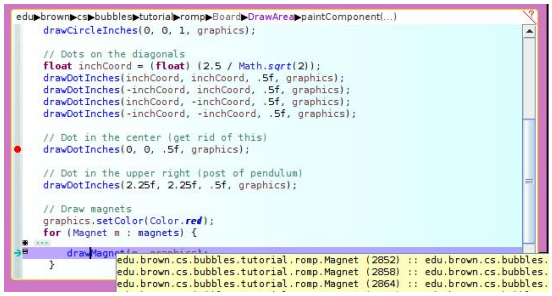
g) Stack view



h) Data view showing variable dependencies



i) File output view



j) Editor view showing highlighted line and tool tips

Fig. 1. Views of the continuous execution display bubble.

that were saved in that variable with a clock indicating when the changes occurred. The cache lets the system access memory at any point in the execution. This structure, while created in the interpreter, is duplicated for variables that are displayed by the front end, allowing the user to see their values over time.

The second component is a file manager responsible for tracking the current state of all active files and building resolved abstract syntax trees for each execution environment as the files change.

The third component is a combination of three interpreters that work off the same value base and the same global clock. The first is an interpreter for abstract syntax trees that is used for code that the user can change. The second is an interpreter for byte codes that is used for library methods as well as parts of the system which are not being edited. The third is an interpreter for native code. This simulates many of the Java library methods that use native code by executing equivalent code in the interpreter or by invoking routines in the debugged process. Strings are handled by this interpreter for efficiency.

The fourth component of SEEDE is a set of output models that reflect the effect of the code on the external environment. The current implementation includes two basic models. One handles graphical output, maintaining the set of graphics commands that are executed for a given window over time. This allows the front end to show the effect of changes on graphical output, something that cannot be done using Java live update. The second is a model of files and the file system. This model essentially provides a shadow file system where the application can read and write without affecting actual files or the environment. Outputs are recorded by time and passed to the front end for appropriate display. The third is a synchronization model that handles locking between threads assuming that all the threads are being interpreted.

The final component is the controller. This understands the debugger session that the user is evaluating from, obtains and caches values from that session for use by the interpreters, tracks multiple threads, handles stopping and rerunning execution when the user makes changes in the editor, handles any interactions between the front and back end (i.e. requests for user console input made by the code or requests to display the result of drawing a particular window after the code has executed), and passes back complete execution information when executions have finished.

V. Limitations

Since the code is being interpreted, and being interpreted potentially on each keystroke, performance of the interpreter can be a major concern. However, since the system is targeted toward developing and debugging a single routine at a particular point in the program, the amount of interpretation may not be that great. Currently we are processing about 150,000 variable updates a second (the interpreter clock ticks each time a value is written). For the examples we have been looking at, response, other than the initial run, has not been a problem.

The second limitation involves determining what the system can and cannot do. There are obvious limits in terms of the interfacing with the outside world when the interpreter is supposed to ensure that nothing external is changed. For example, using sockets to communicate with an external program is problematic. Database access is not handled but could be added with some caveats. Other limits are based on the current prototype implementation. For example, we do not handle all possible file system changes.

Handling synchronization and multiple threads is complicated and SEEDE does not necessarily do it correctly. The problems arise because some of the threads that need to be synchronized might not be simulated. It is difficult to synchronized running threads with the threads being simulated or to detect lock changes in the running threads and propagate them to the threads being simulated. Moreover, the notion of caching values from the running program only works if those values are static.

Other limitations involve what edits can and cannot be supported by the system. Some, such as deleting the routine being interpreted, are difficult to accommodate. Others, such as adding new methods are relatively easy. The interesting ones involve changes that affect the environment before the call. For example, adding a new field to a class requires all objects of that class to have that field and for that field to have a value for those objects. Our current approach handles this but requires either a default or user defined value to be used for all such instances.

VI. Proposed Evaluations

While we originally developed SEEDE to assist in writing new code, we have also noted its potential for debugging. We are currently starting a user study to look at two hypotheses:

- SEEDE helps programmers when writing new code.
- SEEDE can make debugging fast and accurate.

Our user study will look at these two questions. After instructing the participants on the use of Code Bubbles and SEEDE, we will have them do both a debugging and a code writing task. We are considering standard debugging tasks (e.g. from Parnin and Orso [18]) and creation tasks such as the binary search example of Victor [33]. For each task we will have participants do the task using Code Bubbles with and without SEEDE. We will measure time and accuracy of the solutions. We will also get the user's opinions on the utility and appropriateness of the tool.

VII. Availability

SEEDE is integrated into currently available Code Bubbles environment. Code Bubbles is available as a binary distribution from http://www.cs.brown.edu/people/spr/codebubbles. The current source distribution is available from SourceForge. The SEEDE execution engine is available from GitHub.

REFERENCES

[1] Arvind Arasu, Shivnath Babu, and Jennifer Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal* **15**(2) pp. 121-142 (2006).

[2] Steven Arzt and Eric Bodden, "Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes," *2014 International Conference on Software Engineering*, (2014).

[3] R. M. Balzer, "EXDAMS: extendable debugging and monitoring," *Proceeding of the American Federation of Information Processins Societies Spring Joing Computer Conferenece*, pp. 567-580 (1969).

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

[5] Jonathan Edwards, "Example centric programming," *ACM SIGPLAN Notices* **39**(12) pp. P 84-91 (December 2004).

[6] William Finzer and Laura Gould, "Programming by rehearsal," *Byte* **9**(6) pp. 187-210 (June 1984).

[7] Chris Gottbrath, "Reverse debugging with the TotalView debugger," *Cray User Group Conference 2008*, (May 2008).

[8] Muhammad Shams Ui Haq, lejian Liao, and Ma Lerong, "Design and implementation of sandbox technique for isolated applications," *IEEE Informantion Technology, Networking, Electronic and Automation Control Conference*, (May 2016).

[9] Peter Henderson and Mark Weiser, "Continuous execution: the VisiProg environment," *International Conference on Software Engineering 1985*, pp. 68-74 (August 1985).

[10] Taesoo Kim and Nickolai Zeldovich, "Practical and effective sandboxing for non-root users," *Proceedings of USENIX Annual Technical Conference*, (kzpesnru).

[11] Andrew J. Ko and Brad A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," *International Conference on Software Engineering 2008*, pp. 301-310 (May 2008).

[12] H. Lieberman and C. Hewitt, *A Session with Tinker: interleaving Program testing with program Writing, Proceedings 1980 LISP Conference (1980)*.

[13] Henry Lieberman and Christopher Fry, "ZStep 95: a reversible, animated source code stepper," in *Software Visualization: Programming as a Multimedia Experience*, ed. John Stasko, John Domingue, Marc Brown, and Blaine Price, MIT Press (1997).

[14] Kivanc Muslu, Yuriy Brun, Michael D. Ernst, and David Notkin, "Making offline analyses continuous," *ESEC/FSE 15*, (August 2015).

[15] Iulian Neamtiu and Michael Hicks, "Safe and timely dynamic updates for multi-threaded programs," *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13-24 (2009).

[16] Michael Paleczny, Christopher Vick, and Cliff Click, "The Java HotSpot server compiler," in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, , Monterey, California (2001).

[17] Laszlo Pandy, "Elm,s time-traveling debugger," *http://debug.elm-lang.org*, (2017).

[18] Chris Parnin and Alessandro Orso, "Are automated debugging techniques actually helping programmers?," pp. 199-209 in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, (2011).

[19] Fabio Petrillo, Zephyrin Soh, Foutse Khomh, <arcelo Pimenta, Carla Freitas, and Yann-Gael Gueheneue, "Towards understanding interactive debugging," *2016 International Conference on Software Quality, Reliability and Security*, pp. 152-163 (2016).

[20] Guillaume Pothier and Eric Tanter, "Back to the future: omniscient debugging," *IEEE Software* **28**(6) pp. 78-85 (October 2009).

[21] Slowpoke Productions, "Slowpoke Productions," *http://www.slowpokeproductions.com*, (2016).

[22] G. Ramalingam and Thomas Reps, "A categorized bibliography on incremental computation," *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 502-510 (1993).

[23] Steven P. Reiss, "PECAN: program development systems that support multiple views," *IEEE Transactions Software Engineering* **SE-11** pp. 276-284 (March 1985).

[24] Steven P. Reiss, Jared N. Bott, and Joseph J. La Viola, Jr., "Plugging in and into Code Bubbles: the Code Bubbles architecture," *Software Practice and Experience*, (2013).

[25] Steven P. Reiss, "Code bubbles tutorial," *http://www.cs.brown.edu/people/spr/codebubbles/tutorial*, (2015).

[26] Robert V. Rubin, Eric J. Golin, and Steven P. Reiss, "ThinkPad: a graphical system for programming-by- demonstration," *IEEE Software* **2**(2) pp. 73-78 (March 1985).

[27] David Saff and Michael D. Ernst, "An experimental evaluation of continuous testing during development," *Proceedings 2004 ISSTA*, pp. 76-85 (2004).

[28] Alec Sharp, *Smalltalk by Example: The Developer,s Guide,* McGraw Hill (1996).

[29] Andrea H. Skarra, Stanley B. Zdonik, and Steven P. Reiss, "An object server for an object-oriented database system," *Proceedings Workshop on Object- Oriented Database Systems*, (September 1986).

[30] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley, "Dynamic software updates: a VM-centric approach," *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-12 (2009).

[31] Phillip Dale Summers, "Program construction from examples," Yale research report 51 (1976).

[32] Gu Tianxiao, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lu, "Low-disruptive dynamic updating of Java applications," *Inforamtion and Software Technology* ***56***(9) pp. 1086-1098 (September 2014).

[33] Bret Victor, "Inventing on Principle," *Talk at CUSEC 2012. Available at https://vimeo.com/36579366*, (2012).

[34] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook, "Does continuous visual feedback aid debugging in direct-manipulation programming systems?," pp. 258-265 in *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, (1997).

[35] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar, "Native client: a sandbox for portable, untrusted x86 native code," *IEEE Symposium on Security and Privacy*, (2009).

[36] Frank Kenneth Zadeck, "Incremental Data Flow Analysis in a Structure Program Editor," *Ph.D. Dissertation, Department of Computer Science, Rice University*, (1984).