SEEDE: Simultaneous Execution and Editing in a Development Environment

Steven P. Reiss
Brown University
Department of Computer Science
Providence, RI, USA
spr@cs.brown.edu

Qi Xin
Brown University
Department of Computer Science
Providence, RI, USA
qx5@cs.brown.edu

Jeff Huang
Brown University
Department of Computer Science
Providence, RI, USA
ase@jeffhuang.com

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 and data flow, output, and graphics. These results are updated as the user edits. The tool can be used to help the user write new code or to find and fix bugs. The tool is explicitly designed to let the user quickly explore the execution of a method along with all the code it invokes, possibly while writing or modifying the code. The user can start continuous execution either at a breakpoint or for a test case. This paper describes the tool, its implementation, and its user interface. It presents an initial user study of the tool demonstrating its potential utility.

CCS CONCEPTS

· **Software and its engineering → Integrated and visual development environments;** *Interpreters*; Software prototyping;

KEYWORDS

continuous execution, integrated development environments, debugging, live programming.

ACM Reference Format:
Steven P. Reiss, Qi Xin, and Jeff Huang. 2018. SEEDE: Simultaneous Execution and Editing in a Development Environment. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering* (ASE '18), September 3–7, 2018, Montpellier, France. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238182

1 INTRODUCTION

The ability to see intermediate results and understand and check the 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 and Python. For example, Sharp [62] 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 [71]. This notion of *live programming* is slowly spreading to other domains such as data visualization [15,28], and other, generally functional, languages. Programmers can do live coding but only to a very limited extent with today's traditional Java programming environments using hot swapping, the ability to reload a class and continue execution.

Our goal is to provide live programming for real, complex, and long-running systems in a practical fashion. We targeted Java programs as an example, but the techniques could be applied to other languages.

Our prototype tool, SEEDE, provides the ability to see immediately the effect of code changes on execution. It works for complex Java systems and a wide range of different edits. It essentially provides live programming for real programs. It lets the programmer explore the execution both to find problems and to investigate the effect of changes. It does this within the open-source Code Bubbles programming environment [6], letting the programmer start a new session at a breakpoint or for a test case and showing the updated execution as the programmer edits.

Our work offers several contributions including:

- The requirements and insights needed to make live programming possible and practical for real programs.
- A methodology and architecture that implements these insights and meets the requirements.
- An efficient and powerful initial user interface for browsing over the complete execution of a method.
- Techniques for displaying graphical output during execution for both paint methods and components.
- A user study showing that the tool is usable and effective.

The requirements and insights are discussed in Section 2. Related work is given in Section 3. An example use of the system is shown in Section 4. The implementation is outlined in Section 5. The user interface provided by Code Bubbles is described in Section 6. Limitations of the approach are discussed in Section 7. An initial evaluation of the system is presented in Section 8.

2 PRACTICAL LIVE PROGRAMMING

Live programming involves providing continuous execution feedback as the user edits. Any system that does this 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 on the resultant execution within seconds. If feedback is slower, it can either confuse the programmer or cause unnecessary delays.
- *Non-obtrusive.* Execution feedback should not interfere with or require substantial work on the part of the programmer in order to see the results. This is especially true if the results can change on each keystroke.

- *Idempotent.* Continuous execution should not actually change any values in the running program or the external environment. Any such changes would make running the code multiple times problematic. For example, a Java method that starts with the code:
```
  if (!done.add(input)) return;
```
can only be executed once since the next execution would just return.
- *Error Tolerant.* The intermediate code created by the user will contain both syntactic errors and semantic faults. Continuous execution must be able to work in such an environment, providing output at least up to the first error. Moreover, once the user has an error-free program, the displayed execution should be comparable to the prior error-free version.
- *Complete.* The system needs to be able to handle a large fraction of the underlying language. This means being able to handle input-output, files, graphics, as well as all the routines that involve native code. It also means handling a wide variety of edits.

Working with large, potentially long-running programs that interact with the real world complicates the notion of live programming. Making live programming work in this environment requires a number of insights.

The first insight is that, rather than attempting to do live programming over the entire application, it is sufficient to consider only the execution of *a single method and all that it calls.* Programming or debugging a large application is not done all at once. Instead, the programmer will concentrate on a particular feature, a particular bug, or a particular test case. Moreover, if the programmer needs to make significant changes to the overall system, for example doing a major refactoring, live programming is not going to be relevant since there will be few consistent intermediate states between the original and revised system.

For code creation, we assume that the user wants to implement a particular method and needs to get that method working in an appropriate context. This can include adding or modifying other methods, classes, and fields in the system. In this case they would stop either in that method or in one of its callers. For debugging, we assume the programmer has narrowed the problem to the invocation of a single method and all it calls, for example the test driver. Concentrating on a particular method and what it calls addresses the requirements of performance and non-obtrusiveness.

The second insight is that *the execution should be triggered from a breakpoint in a debugger run.* The environment needed to run a method (i.e. all the associated data structures and values) can be large and complex. Associating a SEEDE run with a particular breakpoint in a debugger session lets SEEDE query the debugger to access this environment. The environment would typically be too difficult for the user to specify manually and would take too long to recompute on each keystroke. Starting from a breakpoint means that most of the execution is fixed, enhancing performance. It also allows the use of regular execution to reach the particular situation, for example interacting with other systems or through a user interface, so that these are not continually re-executed.

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. Providing the complete execution simplifies navigation over time for debugging, making the result non-obtrusive since the programmer will not have to do that much work to get back to the state they last looked at before an edit.

Java already has a form of live programming through the ability to hot-swap code in most modern debuggers [43]. To use this feature, the user edits the code and then saves it. Execution then goes back to the start of the current routine and the user can proceed normally. While this is helpful, it is often problematic in practice as discussed below. One of our goals was to provide an implementation framework that can provide the benefits of hot swapping without the problems.

The first problem is that the hot-swapping operation often fails. The code has to be error-free. The changed code cannot add or remove any fields or methods or change any method or field signatures. Any routine that has changed can not be executing in another thread or recursively in the current thread. Changes to static field values are not propagated. Such failures are common when trying to use hot-swapping. Moreover, once hot-swapping fails, it is difficult to continue. Our approach handles all these changes, either automatically or with minor user intervention (for example when a new field needs a non-default value for existing objects).

The second problem is that the point where execution continues changes from the current stopping point to the start of the method, or, in some cases, the start of some prior method. The user has to step or continue the execution to the point of interest. Our approach tracks the current position in the execution and automatically restores it after an edit.

A third problem is that hot-swapping is not idempotent. If execution of the function makes a change to the environment or parameters during one execution, that change persists when the execution restarts, making the re-execution different. We deal with this by executing outside the actual environment.

A fourth problem is that hot-swapping execution affects the outside environment. It re-executes external I/O each time, causing multiple occurrences of output and requiring the user to reenter input each time. We address these through an I/O model for the console and files. Hot-swapping also has problems with synchronization. For example, doing a hot swap while holding a lock does not release the lock. This is true both for Java synchronization locks and for user-defined locks. With our approach, locks are only maintained within the simulation, not in the original program.

A fifth problem is that hot-swapping cannot be used easily to work on problems involving graphics. Stopping in a paint routine implies stopping the graphics thread. Changes to graphics are not visible until the graphics thread resumes and control returns to the system routines. Changes to a routine setting up a widget cannot be seen until the overall window has been setup and the graphics thread runs. Using a graphics model, our approach is able to show the intermediate graphics results for both these cases.

3 RELATED WORK

The idea of providing immediate execution feedback while coding was central to spreadsheet programming introduced by Visi-

Calc [7]. The idea was picked up for more traditional programming by VisiProg [25], and more recently in the EG extension to Eclipse [16]. 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 [71]. Since then, there have been numerous attempts at providing similar functionality, including an experimental mode of the Python Tutor [22], Choc [40], JSFiddle [27] and Schuster's work [59] for JavaScript, Eve [38], Lambu for Haskell [32], Unison [12], and Liveweave [61] and Chickenfoot [66] for web applications, among others. Most of these are designed for simple programs, not large existing systems.

A more extreme version of live programming involves using examples to create code directly. This can be seen in the various programming-by-example systems that have been developed over the years [17,56,68]. More recent examples combine examples with live programming [21,60]. The approach is also being used effectively for database interactions using continuous queries [1] and in interactive data exploration tools [15,28].

There have been several studies on how programmers debug and on what tools and techniques might be helpful for debugging [45,47,73]. These tend to show that the type of 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 [35], and more recent efforts involving continuous testing [57]. The work is also related to incremental execution [36,51] and continuous and incremental program analysis [2,41,50,75].

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 [51]. The algorithm animation system BALSA provided a time slider similar to the one we offer [8]. More recent debuggers that include similar features include TotalView [20], Elm's time-traveling debugger [44], and the Trace-Oriented Debugger [48]. Ko's Whyline provided similar capabilities in a question-answering framework [31]. Recent interest in this area involves time-travel or omniscient debugging [4,10,13,14,23,26,30,33,72].

Dynamic updating of data structures has been used for maintaining long-running applications. This involves taking updates and modifying the existing system to use the new code [42,67,70]. These require the programmer to identify safe points for update and concentrate more on migrating object implementations from one version to the other. While some of these technologies are useful, most of them are 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 [63]. Our approach uses appropriate techniques from these system to simulate object migration where necessary.

The use of models for simulating the external environment is called mocking and is used in some testing environments [19,37,39,46,64,69]. Sandboxing of files is also used extensively for security purposes [24,29,74].

4 EXAMPLE USE

In this section we give an example of the use of SEEDE for debugging. We note the tool can also be used for writing new code in much the same way.

The Code Bubbles tutorial program [54] is a simulation of the Romp toy [49] consisting of a pendulum with a magnet moving chaotically over a surface with movable magnets. 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. This is difficult to do in the debugger because changes to the graphics parameters are not immediately visible and the exact changes are not obvious. To use SEEDE on the 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. Then we right click to bring up the default pop-up menu, and select "Start Continuous Execution".

At this point SEEDE starts the continuous execution process. After about 5 seconds (the initial run is slow because code and debugger data needs to be loaded), the system populates the various components of the continuous execution bubble with the resultant values as shown in Fig. 1a.

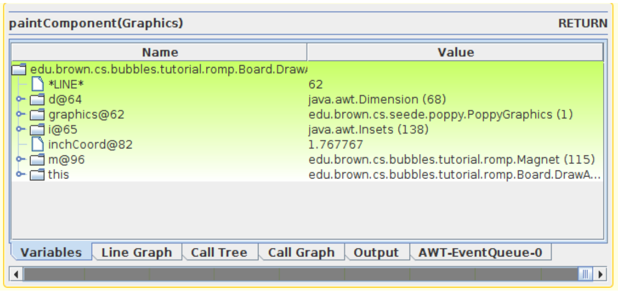
The default view provided is a hierarchical view containing variables and their values from the simulated execution, similar to a debugger display. The scroll bar at the bottom of the display lets the user scroll over time in the execution. Green areas in the scroll bar represent code in the current method; gray areas represent code in called methods. As the user scrolls, the variable values change accordingly, and the displays at the top left corner are updated to show call stack. A special variable, *LINE*, shows the current line number at that point. This line is also highlighted in any editor that is open and includes the method as seen in Fig. 1h.

For our example, since we are concerned with drawing the magnets, we slide the time scroller to the gray area representing one of the *drawMagnet* calls, and then right click to select that context. This could also have been done using the call graph (Fig. 1b) or stack (Fig. 1e) view. We next right click again to have SEEDE bring up the corresponding source.

Since the top level routine is a paint routine, SEEDE includes a graphics panel showing the paint result as computed by interpreter in Fig. 1c. We find the code that sets the color to green and change the reference to red. Within half a second the graphics output changes accordingly. Changing the offset for the '+' and '-' tokens is more complex. We find the *drawString* calls and try out different delta values. As before, immediately after we have set a new value, we can see the result. We eventually find the proper deltas for the '+' token and then repeat the process for the '-' token. The final results can be seen in Fig. 1d.

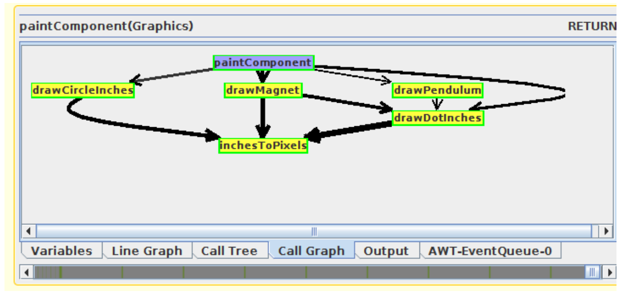
5 IMPLEMENTATION

SEEDE runs as a separate process, talking to both an interface within the Code Bubbles system (BICEX) and to Code Bubbles' Eclipse-based back end through a messaging interface [53] as seen in Fig. 2. It takes requests from Code Bubbles and sends execution updates back to it asynchronously as they become available. It uses the back end to query the values of variables, to understand the Java environment, and to detect changes both to the execution and to files being edited.

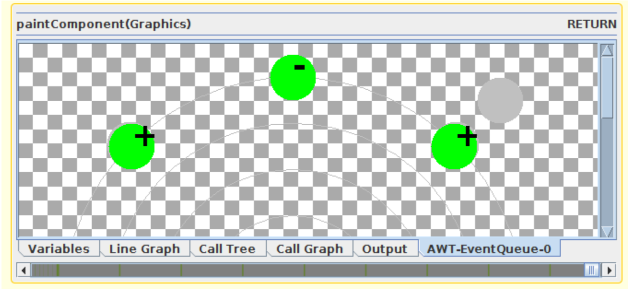

paintComponent(Graphics) RETURN

Name	Value
edu.brown.cs.bubbles.tutorial.romp.Board.Draw/	
LINE	62
d@64	java.awt.Dimension (68)
graphics@62	edu.brown.cs.seede.poppy.PoppyGraphics (1)
i@65	java.awt.Insets (138)
inchCoord@82	1.767767
m@96	edu.brown.cs.bubbles.tutorial.romp.Magnet (115)
this	edu.brown.cs.bubbles.tutorial.romp.Board.DrawA...

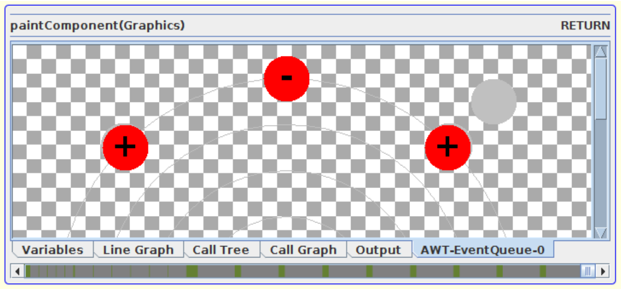
a) Variable view showing time slider and values



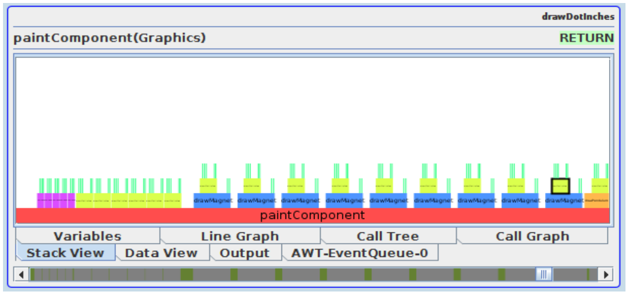
b) Call graph view of execution



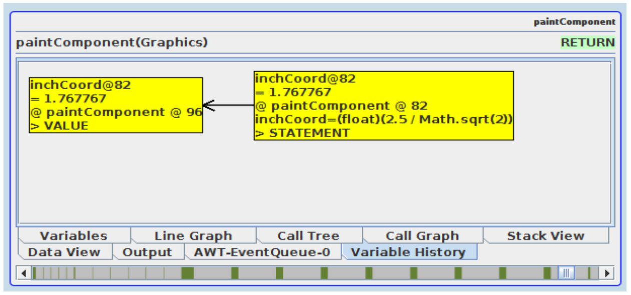
c) Initial graphical output window



d) Graphical output after editing



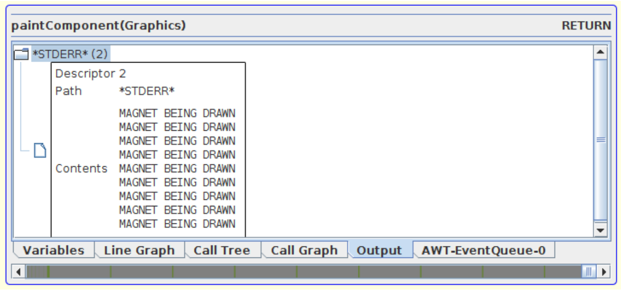
e) Stack view



```
inchCoord@82
= 1.767767
@ paintComponent @ 96
> VALUE
```

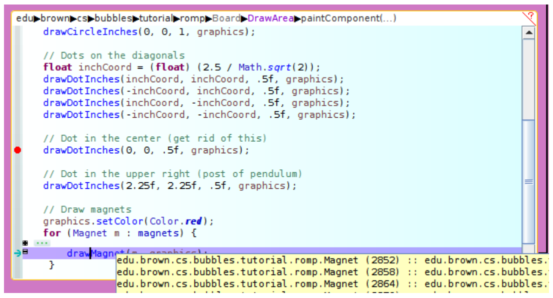
```
inchCoord@82
= 1.767767
@ paintComponent @ 82
inchCoord=(float)(2.5 / Math.sqrt(2))
> STATEMENT
```

f) Data view showing variable dependencies


paintComponent(Graphics) RETURN

```
*STDERR* (2)
Descriptor 2
Path       *STDERR*
           MAGNET BEING DRAWN
           MAGNET BEING DRAWN
           MAGNET BEING DRAWN
           MAGNET BEING DRAWN
Contents   MAGNET BEING DRAWN
           MAGNET BEING DRAWN
           MAGNET BEING DRAWN
           MAGNET BEING DRAWN
           MAGNET BEING DRAWN
```

g) File output view


edu▶brown▶cs▶bubbles▶tutorial▶romp▶Board▶DrawArea▶paintComponent(...)

```
    drawCircleInches(0, 0, 1, graphics);

    // Dots on the diagonals
    float inchCoord = (float) (2.5 / Math.sqrt(2));
    drawDotInches(inchCoord, inchCoord, .5f, graphics);
    drawDotInches(-inchCoord, inchCoord, .5f, graphics);
    drawDotInches(inchCoord, -inchCoord, .5f, graphics);
    drawDotInches(-inchCoord, -inchCoord, .5f, graphics);

    // Dot in the center (get rid of this)
    drawDotInches(0, 0, .5f, graphics);

    // Dot in the upper right (post of pendulum)
    drawDotInches(2.25f, 2.25f, .5f, graphics);

    // Draw magnets
    graphics.setColor(Color.red);
    for (Magnet m : magnets) {
       drawMagnet(m, graphics);
    }
```

```
edu.brown.cs.bubbles.tutorial.romp.Magnet (2852) :: edu.brown.cs.bubbles.
edu.brown.cs.bubbles.tutorial.romp.Magnet (2858) :: edu.brown.cs.bubbles.
edu.brown.cs.bubbles.tutorial.romp.Magnet (2864) :: edu.brown.cs.bubbles.
```

h) Editor view showing highlighted line and tool tips

Figure 1: Views of the continuous execution display bubble. The main view (a) shows the variables in the current context. The methods used in the current context are shown in a call graph (or tree) view (b) or in a stack view (d). Graphics output for underlying or user selected component can be displayed (c and d). Variable dependencies for a user selection is available (f), as is the output to files or the console (g). The interface also coordinates with the Code Bubbles editor (h).

While SEEDE could be made to work with any existing programming environment, we found it simplest to make the prototype work with Code Bubbles. Code Bubbles made it easy to create a new display that did not interfere with or take space away from existing displays. It simplified determining what files the programmer is working on since it maintains the programmer's current working set including all relevant source code editors. The existing message interfaces made it easy to implement the system as a separate process which facilitated debugging and experimentation, and ensured that the system did not interfere with the actual environment.

SEEDE consists of five main components. The first is a controller that handles all communication and starts and stops the simulated execution as needed. The second is a file manager that maintains the current contents of all active files and updates their abstract syntax trees as they change. The third is a set of three

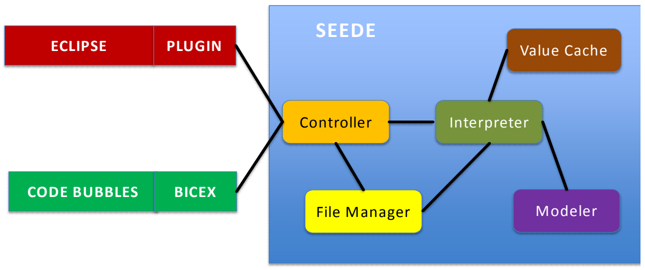


Figure 2: Overview of the SEEDE architecture. SEEDE runs as a separate process that talks to both Code Bubbles and Eclipse. SEEDE itself consists of 5 components. The controller manages the communications and execution. The file manager tracks the active files and handles recompilation. Three interpreters are combined to do the simulation, one for editable code, one for compiled code, and one for special cases. The value cache maintains values over the whole run. The modeler provides models of graphics, files, synchronization and I/O to mimic the external environment.

interpreters, one for abstract syntax trees to handle code that is or could be edited, one for byte code to handle library and other static code, and one to handle native methods and special cases. These interpreters utilize the fourth component, a value cache, that tracks all values over time. The final component is a suite of models that reflect the external environment.

The SEEDE architecture is engineered to meet the requirements of Section 2. The architecture is designed to be fast enough to re-execute as the user types. It is capable of restarting execution on each edit. It minimizes the amount of information that needs to be sent back for display purposes. The implementation is a simulation that starts with the environment at a breakpoint and values from the debugger are cached. Values are then only changed in the simulation, ensuring idempotency. The simulator uses the Eclipse parser to generate valid abstract syntax trees in the face of errors and detects both syntactic and semantic errors that need to stop execution. Finally, by effectively recompiling and reexecuting all changed files, it can handle a wide range of edits including adding new fields, variables, methods, classes, etc. The use of models enables it to handle I/O and graphics operations within the simulator without affecting the external environment. Details of the essential design decisions are outlined below.

5.1 The Controller

The first component is the controller. The controller monitors messages both from Eclipse (via the Code Bubbles plugin) and from Code Bubbles itself to maintain a set of execution environments each defined by a stopped debugging run. The controller uses the message interface to determine the threads that are stopped and creates an internal execution for these threads without any additional user input. It assumes that the execution has stopped at the beginning of the stopped function(s). The system still works if this assumption is not valid (i.e. the user stopped the method with a breakpoint in the middle) provided that none of the values relevant to the execution of the function was yet affected (i.e. the function is idempotent up to the stopping point).

For each execution environment, the controller tracks the set of files that are active and controls the simulated execution. The set of active files is maintained automatically by BICEX by monitoring changes to the user's working set that contains the SEEDE display, a functionality provided by Code Bubbles. When the execution completes, the result is sent asynchronously via the messaging interface to Code Bubbles and the displays are updated accordingly.

The controller monitors messages from Eclipse indicating that a file has been edited. If the file is one of those associated with a simulated execution, that execution is stopped if it is running, the changed file is updated, the project is recompiled using the change, and the execution is restarted. Multiple edits are grouped when possible.

The controller handles requests from Code Bubbles, including changing the initial value of variables, adding variables to be viewed as graphical components. removing executions when they are no longer needed, expanding the set of sub-values of a value, and returning the dependencies of a variable at a particular location.

5.2 The File Manager

The file manager is responsible for tracking the current state of all active files and building resolved abstract syntax trees for each execution environment as the files change. Compilation is handled using the Eclipse Java parser to build abstract syntax trees, and the fast and forgiving compiler from the open-source S^6 system [52] to resolve these trees against other current files and the rest of the project.

5.3 The Interpreters

The next 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 to handle special cases such as native methods.

The abstract syntax tree interpreter uses the resolved abstract syntax trees generated by the file manager. This interpreter is error tolerant and lets the simulation work without the need to save files or do code generation which could be expensive. The byte code interpreter uses ASM [9] to load class files from the class path. This interpreter lets SEEDE run real programs that use libraries without their source and to efficiently execute those parts of a larger program the user is not working on. Both interpreters update the special variable *LINE* to show the currently executing line.

The special case interpreter is used to handle code that cannot be directly interpreted, for example native methods, to deal with code that affects the outside world, and to make the overall process more efficient. Several classes, notably *String*, *Class*, and *File* are handled as internal objects rather than run-time objects. Strings are handled this way for efficiency; classes are handled because the interpreter has its own type model that needs to be updated dynamically; files are handled to accommodate the file model described in Section 5.5

The special case interpreter runs in place of the byte code interpreter for specific classes and methods. Where it is possible, for example the various native math and string routines, the method is handled by appropriate code in the interpreter itself. In some cases, for example hash codes, the system will query the debugger for an appropriate value. For other cases, the interpreter invokes methods in a small library SEEDE installs in the user's process. This is used, for example, to get information about resources from a class loader.

All three interpreters utilize a common clock that increments with each write, a common run time stack for each interpreted thread, and a common context for looking up variables by name or reference. The choice of which interpreter to use is determined at the start of each method call.

5.4 The Value Cache

The fourth component in the SEEDE framework is a value cache. One of SEEDE's innovations is that it returns the complete evaluation. The value cache is designed to facilitate this by maintaining all values over time. A simplified version of the cache is then sent to Code Bubbles where it is used as the basis for the various displays.

The value cache starts with a set of base values which can represent the Java primitive types, the types that are managed internally (String, Class, and File), and generic types for arrays and objects. The cache also defines a reference value which is a map that yields the actual value of the reference at any particular time. Each variable in the interpreter, as well as each field of an object and each element of an array, is assigned such a reference value.

In addition to these explicit values, the value cache provides latent values that can refer to arbitrary expressions or nested variables in the debugger. When a value from the debugger is accessed, the information returned only includes the top-levels of any nested structures. Any deeper references are replaced with latent values. Latent values are only computed when they are needed by the execution and are not reported to Code Bubbles. This can greatly reduce the amount of information needed from the debugger and passed to Code Bubbles and thus significantly speed up the overall execution process. Users can manually request additional values to be displayed for a variable or change the initial value of a variable retrieved from the debugger.

The value cache also tracks the unique identity of each object so that two references to the same object actually point to the same value. Reference information is passed to Code Bubbles, again reducing the amount of data that needs to be communicated. Another similar optimization notes what values have not changed from one run to the next and passes a reference to the value in the previous run.

5.5 The Models

The final component of SEEDE is a set of models that reflect the effect of the code on the external environment. These let SEEDE simulate and report external changes without actually affecting the environment. SEEDE currently supports a file model, an input-output model, a graphics model, and a synchronization model. Other models, such as one supporting database operations, can be added.

The file model provides a simulated file system where the program can create and remove files and change file properties without actually affecting the external environment. This lets the program perform file operations without any side effects. The simulated file system can be quickly restored to the initial configuration.

The input-output model tracks the contents of reads and writes done by the interpreted code. It returns any writes done to files, including standard output and standard error, to the front end along with the time of the write so that the corresponding output can be displayed. It maintains the original file position of each file being read so that read operations will be idempotent. It also maintains a console input buffer. The first time the program requests console input, the user will be prompted accordingly. When the program is rerun, say after an edit, the saved input will be reused.

The graphics model provides a means for displaying or understanding the graphics operations that are done during an execution. This is implemented by replacing any *Graphics* parameter in a paint routine with a special class based on Java's *DebugGraphics* class that records all operations and the times of each operation. The result is passed to Code Bubbles so that the effective graphics display at a given time can be recreated and shown to the user. This is done automatically for the various Java paint routines.

The graphics model can also be used to look at graphical widgets that are built or modified in the simulated code. This is done by having the user select the widget and request a graphics display. The simulator then, once the overall run is finished, simulates a separate call to paint on the selected widget using the special *Graphics* class and returns the result for display.

The synchronization model is used to simulate Java locks among the threads that are being interpreted. If there is only one such thread, this model is ignored. The model does not currently handle locking with respect to running threads in the application that are not being interpreted.

6 USER INTERFACE

The user interface for continuous execution, shown in Fig. 1, is key to making the tool both usable and useful. The interface can be initiated in one of two ways. First, there is a menu button to "Start Continuous Execution" which finds an execution stopped at a breakpoint and sets up SEEDE for that execution using the editors in the current working set. Second, the user can right click on a test case in the test management bubble and select "Show Execution". This automatically creates a debug session for the test case, sets a breakpoint at the start of the test, runs the test case up to that breakpoint, removes the breakpoint, and then starts up SEEDE. In this case, Code Bubbles automatically determines all the project code that is used in the test case using its coverage tool and indicates that these sources should be viewed as editable and hence interpreted using the abstract-syntax tree interpreter and included explicitly in the output display.

The user interface is designed to display the full execution of the stopped method and everything it calls using multiple views in a way that simplifies navigation and exploration while being unobtrusive. It is designed to help the programmer focus on a particular call while allowing easy exploration over time both within that call and over the whole execution. It maintains the current context the programmer is focused on, including the display and the time, and restores it after a change. It provides a simple scroll bar at the bottom to navigate over time within the execution. All views are synchronized to this scroll bar.

The user interface shows a single invocation of a single method, automatically breaking the overall execution into executions of individual methods. This helps the programmer focus and minimizes the amount of display space needed so the user has full access to the rest of the programming environment while exploring. This also ensures that the user does not lose context when scrolling over time. The calling context of the displayed method is shown at the top of the window.

The interface provides multiple views of that single invocation. The main view, shown in Fig. 1a shows the values of variables in the current invocation at the selected time. This is a hierarchical display similar to that provided by the debugger. Local variables are identified with their declaration line number to avoid confusion when multiple variables with the same name exist in the method.

The user interface currently only shows routines that are interpreted based on source code, i.e. with the abstract-syntax tree interpreter. System routines and parts of the system that the user is not currently editing are omitted. This keeps the display and the set of contexts at a reasonable size. The user can open a new editor to cause the execution to be reinterpreted and the corresponding code to be included in the display.

Because only one invocation or context is shown at a time, it is essential to the interface that the user be able to navigate to other relevant contexts easily and quickly. Generic navigation commands let the user go to an inner context at a particular time, go to the next or previous invocation of the method, go to the next or previous line, go to the next or previous time the current line was executed, or go to the parent context. The overall bubble includes breadcrumbs at the top to let the user go to any particular parent context of the current context.

Additional views are provided primarily to aid in navigation. The call graph and call tree views show the method calls as in Fig. 1b. These differ in that the call tree contains a node for each invocation whereas the call graph contains one node for each method. The user can right click on a node here to change the current context to one corresponding to the displayed function. With the call tree, the context will change to the given invocation. For the call graph, the context will be the first invocation of that method from the current context. The stack view of Fig. 1e shows a compact display of the stack over time within the context with different called methods color-coded similar to [55]. In the stack view, the user can select a particular inner context to become current.

The user can also navigate using data flow. One way of doing this is to right click on a variable value and go to the time and context where that value for the variable was set. An alternative is to select a variable value and request the data flow dependency graph leading to that value. This is computed mainly by the SEEDE back end upon request and results in a display showing dependencies as in Fig. 1f that can then be used for navigation.

Two additional views provide access to the input/output and graphics models computed during the execution. The first shows all file outputs, organized by file. The text output displayed for each file is synchronized with the current time as shown in Fig. 1g. The second shows a representation of graphics output over time. Examples of this are shown in Fig. 1c and Fig. 1d.

The user interface of SEEDE is also coordinated with editors in the programming environment. The user can request a view of the current method or a method selected in the call graph, and a corresponding editor will be created nearby on the display. The current line being viewed in the SEEDE display will be highlighted in any editors displaying that line. The highlighting changes dynamically as the user uses the time scroll bar or other navigation techniques. Finally, hovering over a variable active in the SEEDE execution in an editor will display a textual view of the value history of that variable as a tool tip. This can be seen in Fig. 1h.

7 SYSTEM LIMITATIONS

Since the code is being interpreted potentially on each keystroke, performance can be a major concern. However, since the system is targeted toward developing and debugging a single routine and everything it calls at a particular point in the execution, 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 system also currently includes a time out after about a minute (10,000,000 writes) to handle particularly long-running computations or accidental infinite loops.

The initial run involves loading all the necessary binary files from the class path and loading all variables accessed by the code that are not explicitly defined in the run from the debugger. An initial run on a complex system, either one that involves a large amount of library code, or that involves large or complex data structures that are actually accessed, can take time. The maximum we have seen in our examples is about one minute. Most examples we have seen, however, complete the initial run in ten seconds or less.

The second limitation involves what the system can and cannot do. There are obvious limits in terms of the interfacing with the outside world when the interpreter must ensure that nothing changes. For example, using sockets to communicate with an external program is problematic. Other limits are based on the current prototype implementation. For example, we currently do not handle jar and zip native methods or random access files. The focus on the execution of a single routine means that edits that would affect the initial environment are not reflected in the output. We provide the user with the ability to change the initial environment to accommodate this.

Handling synchronization of multiple threads is complicated and SEEDE does not necessarily do it correctly. The problems arise because some of the threads being synchronized may not be included in the simulation. It is difficult to synchronize running threads with the threads being simulated, to detect lock changes in the running threads and propagate them to the threads being simulated, or to continually and consistently propagate changing values from the running program. Currently, SEEDE simulates locks and values correctly between simulated threads and ignores threads that are still running. Another problem with multiple threads that we currently ignore is that the result could be nondeterministic and could change on each edit.

There are also limitations in the user interface. Providing only one frame at a time in the stack might be confusing to those accustomed to seeing the complete call stack in the IDE debugger. Our approach is designed to facilitate quickly changing time (for example using the time scroll bar) without causing the programmer to lose context. Another problem with the interface is that we only present values that are actually used in the execution. If the programmer is used to an IDE where they can look at arbitrary values and through arbitrary nestings, they might miss this capability. To accommodate this, we have added specific user commands to expand the presented values on demand. The user interface also does not currently provide the *toString* output for each value, making the variable display less useful than the default debuggers. Other potential user interface limitations, such as the size of the scroll tabs, the unintuitiveness of colors in the scroll bar, and the difficulty in understanding and using the control flow navigation views, were pointed out in the user study.

8 EVALUATION

We did a user study with 29 participants to evaluate the effectiveness of using SEEDE for software development and debugging. In the study, a participant first reads a tutorial to get familiar with the Code Bubbles environment and learn how to use SEEDE. Then he/she does two tasks: a development task (Task 0) and a debugging task (Task 1). To implement a controlled experiment, a participant was asked to use SEEDE to do only one of the two tasks (chosen at random). This way, for each task, we had participants who did the task using and not using SEEDE, and we compared their performance. For each task, we asked questions, collected the participant's created program, and recorded how long it took the participant to finish. We determined whether the participant successfully completed the task by the answers they gave and the program they created. After the tasks, we asked each participant to complete a survey concerning (1) the participant's background, (2) the usefulness of SEEDE, (3) whether they liked SEEDE, and (4) how to improve SEEDE.

Our results show that (1) overall participants felt positive about the utility of SEEDE for the task they did and for programming tasks in general; (2) overall participants liked using SEEDE; (3) SEEDE markedly helped participants succeed in the debugging task; and (4) SEEDE helped participants succeed in the development task, though this was not statistically significant.

8.1 The User Study

We recruited 29 participants from among computer science students at Brown University who had taken at least one course on object-oriented programming. Of the participants, 4 were graduate students and 25 were undergraduate students. Our survey results show that 27 out of the 29 participants had taken at least one additional advanced programming course.

In the study, a participant first reads a tutorial on how to use Code Bubbles and SEEDE and then does two tasks: a development task (Task 0) and a debugging task (Task 1). We randomly assigned a participant to either Group A or Group B. This random assignment yielded 13 participants in Group A and 16 participants in Group B. We asked participants in Group A to do Task 0 using SEEDE and do Task 1 without using SEEDE. We asked participants in Group B to do the opposite, i.e., to do Task 1 using SEEDE and do Task 0 without using SEEDE. We evaluated SEEDE by (1) comparing the *performance* of the participants in Group A with those in Group B for each task; (2) asking each participant (in the survey) whether he/she agrees that SEEDE was useful for doing the task (we asked the participant to choose a score from 1 to 5 where 1 represents *Strongly Disagree* and 5 represents *Strongly Agree*); (3) asking each participant whether he/she agrees that SEEDE was useful for doing a programming task in general (using scores from 1 to 5); and (4) asking each participant whether he/she likes SEEDE (using scores from 1 to 5 where 1 means *I do not like it at all* and 5 means *I like it very much*).

The performance of a participant was evaluated by *success*, i.e., whether the participant succeeded the task, and *efficiency*, i.e., how long it took the participant to finish the task. We evaluated the success of a participant by checking whether the participant correctly answered the questions associated with the task; whether the program created by the participant passed the test suite; and whether the program created by the participant was correct in general and did not over fit to the test suite. For the latter, we manually checked the code the participants wrote.

After the study, we surveyed each participant, asking them to provide their background, to rate their programming skill (in five levels from beginner to expert), to indicate the courses they had taken, and to provide feedback on whether the tutorial was useful, whether SEEDE was useful for doing the task they did and for doing a programming task in general, whether they liked using SEEDE, and how SEEDE can be improved.

Each recruited participant did the study on a computer with 4 AMD Phenom II X4 955 cores and 16G memory where the Code Bubbles environment and SEEDE were previously installed. We asked each participant to finish the tutorial within 30 minutes and to finish each task within 40 minutes. (We did a pilot study with another three participants to determine these times. These participants did not participate in the formal study.)

8.1.1 Tutorial. The goal of the tutorial is to teach a participant how to use Code Bubbles and SEEDE. More specifically, it taught a participant (a) how to open, move, and close a bubble of code fragment (e.g., a method); (b) how to run the program against the test suite; (c) how to invoke SEEDE; (d) how to use SEEDE to see and understand the continuous execution results; (e) how to navigate within SEEDE to show the continuous execution results for different methods; and (f) how to use the Code Bubbles' debugger without using SEEDE.

In the tutorial, a simple task was used to teach the above skills. The task was to write code to complete the development of the method *pop* for a program called *ArrayStack* (153 LOC) which implements a stack structure using an array. The incomplete *pop* method contains only one line as *return null;*. The participant should create a correct implementation for it such that the completed program can pass all the test cases one of which the original, incomplete program failed. By reading the tutorial, the participant was guided to finish the task step by step. The participant was guided to first write code to create a buggy version of the *pop* method. Next the participant was guided to fix the bug to finish the task in two ways: using SEEDE and using the Code Bubbles debugger without SEEDE.

8.1.2 Task0 (development task). In this task, a participant needs to write code for the method *join* in a Java class named *MergeSort*. The original version of *MergeSort* (72 LOC) is from Chapter 12 of Savitch's textbook [58] where the method *join* (40 LOC) accepts as input four parameters: an array *a* of doubles, a starting index *begin*, a splitting index *splitPoint*, and an ending index *end*. On input, the array elements from *begin* to *splitIndex* (inclusive) and the array elements from *splitIndex+1* to *endIndex* (inclusive) are both sorted in ascending order. Given the four parameters, *join* changes the elements of *a* such that the elements from *begin* to *end* (inclusive) are sorted in ascending order.

Table 1. Results of the User Study

Task	Group A (13)			Group B (16)		
	Success #(%)	FoundFault #(%)	Time (min.)	Success #(%)	FoundFault #(%)	Time (min.)
Task 0	6 (46.2)	N/A	29.5	7 (43.8)	N/A	23.1
Task 1	2 (15.4)	5 (38.5)	21.5	7 (43.8)	12 (75)	20

Table 2. Survey Results from the User Study

Group (#Participants)	SEEDE		
	Usefulness for Task	Usefulness in General	Degree of Favor
A (13)	4.2	3.8	3.7
B (16)	3.4	3.9	3.6
A&B (29)	3.8	3.8	3.7

For the development task, we created 10 test cases: 7 to test the method *join* directly, and 3 to test the sorting method from which *join* is called. The original program passed all the test cases. To create an incomplete method for the development task, we removed all the original code from the method body of *join*. The incomplete program only passed one test case. The goal of the task is to create a correct implementation for *join* such that the created program can pass all the test cases. Participants in Group A were required to use SEEDE for doing the task. Participants in Group B were required not to use SEEDE.

It is often challenging to design a development task for a user study that is not too easy but can still be done within a reasonably short amount of time (e.g., 40 minutes). We chose the method *join* from a mergesort program for the task because (1) the input parameters and the expected semantics of *join* are easy to understand, and (2) writing code for *join* is not trivial but still not too difficult. The participants of our pilot study all made mistakes in their coding but still finished the task within 40 minutes.

8.1.3 Task1 (debugging task). In the debugging task, a participant needs to locate and fix a real bug (id: *MU_AK_1*) contained in the *gcd* method (59 LOC) of the Apache Commons Math project (94,609 LOC). The bug was chosen from the 26 bugs used in [34] for evaluating a user-interactive fault localization approach. The bug caused an integer overflow failure and is exposed by a failed test case associated with the project. A correct fix to resolve the failure, according to the developer patch for this bug, is to change the if-condition in *gcd* from `u*v == 0` to `u==0 || v==0`.

Debugging is in general laborious and time-consuming. A person can take hours, sometimes even days, to fix a real bug. We chose the bug *MU_AK_1* for the debugging task because (1) the fix is relatively simple, and (2) the expected semantics of *gcd* is generally known: it computes the greatest common divisor of two numbers. In the task, a participant (from either Group A or Group B) was told that the bug was in the method *gcd*. This makes it possible for a participant to do the debugging task within a relatively short amount of time. (Two of the three participants in our pilot study successfully did the task within 40 minutes.) At the same time, locating and fixing this bug is not trivial since the implementation of the method *gcd* is not the common, one; it is based on an algorithm described in [65] and is 59 LOC.

The project containing the bug has thousands of test cases. It takes about three minutes to run all these tests. This makes a participant wait too long to check a sequence of fixes. To mitigate the problem, we only used the 52 test cases for the failed test class (*MathUtilsTest)* and removed all others. For the task, the participant was informed that there is a bug in the gcd method exposed by a failed test case. We did not provide any more information (e.g., which statement in *gcd* contains the bug). The program fixed by the participant should pass the test cases and be valid in general (determined by our manual examination). In the task, participants in Group B are required to use SEEDE while participants in Group A are required to not use SEEDE.

8.2 Results

Table 1 shows the study results. It lists the number of participants in each group who succeeded in each task, the number that found the fault for Task 1, and the average time it took for the participants to perform each task.

Table 2 summarizes the survey results.. The two usefulness questions used the values 1-Strongly disagree, 2-Disagree, 3-Neutral, 4-Agree, and 5-Strongly agree. The degree of favor question used the values 1-Does not like it at all, 2-Does not like it very much, 3-Neutral, 4-Like it, and 5-Like it very much. As seen in the table, we found that overall (1) the participants feel positive about the utility of SEEDE for the task (Task 0 for Group A and Task 1 for Group B): the usefulness score on average is 3.8 (above neutral and close to useful); (2) the participants also feel positive about the utility of SEEDE for a programming task in general: the usefulness score on average is 3.9 (above neutral and close to useful); and (3) participants liked to use SEEDE: the score for this question is 3.7 (above neutral).

We found that the participants liked seeing the continuous execution results that SEEDE creates. In the survey, we asked each participant the question: If SEEDE was useful, how did it help? And we got many answers similar to the following:

"Being able to scroll through the execution process was very intuitive, and helped me track variables very easily."

By comparing the performance of participants between Group A and Group B for each task, we found that SEEDE markedly helped participants in Group B during the debugging task (Task 1), although this was not statistically significant (Fisher's Exact Test [18], p=0.13) due to the small numbers of completed tasks.). As shown in Table 1, 7/16=43.8% of the participants from Group B successfully did the task using SEEDE, but only 2/13=15.4% participants from Group A succeeded without using SEEDE. The time lengths used by the two groups for finishing Task 1 are comparable. A participant who failed the task still might have successfully identified the location of the bug, and we asked each participant explicitly to identify the faulty code. 12/16=75% participants from Group B successfully identified the bug location using SEEDE, but only 5/13=38.5% participants from Group A did so without using SEEDE.

In the program of Task 1, when the two integers u and v take the value *3145728* and *5242880*, the if-condition $u*v$ evaluates to *0* because of integer overflow. For this task, it is not easy for one to identify and fix the bug without actually looking at the run-time values. By using SEEDE, one sees the execution of the buggy method *gcd* in a continuous way. At each step, the executing statement is highlighted. By viewing the highlighted statements in a continuous way, a participant can identify the execution path of *gcd* with the failure-exposing input, and easily find that the else-branch of the if-statement whose condition is `u*v == 0` is taken even though u and v both have non-zero values. This is very suspicious. Using a debugger is more difficult in this case. A participant needs to set up a breakpoint first and goes step by step to examine the values. The debugger only allows a participant to see the execution for one step. If the participant misses a step, he/she cannot go backwards in the debugging run and might need to start over.

Comparing the performance of participants on Task 0 (the development task), we see that SEEDE helped participants from Group A to succeed in the task, but this result is not statistically significant: 6/13 = 46.2% participants from Group A succeeded using SEEDE, and 7/16=43.8% participants from Group B succeeded without using SEEDE. Though SEEDE did not significantly help participants in Group A succeed in Task 0, our survey results show that participants found SEEDE to be useful during the task: the average score they gave for measuring the usefulness of SEEDE for this task is 4.2 (4 means a participant agrees that SEEDE was useful). These results are consistent with previous findings on the effectiveness of live programming [5,11].

Based on the results, we believe SEEDE was actually useful for this task, but whether a participant can succeed or not largely depends on his/her programming ability with respect to the complexity of the problem. A good participant can develop a *join* method quickly that is either correct or nearly so and does not need SEEDE for any help. A weaker participant may make many mistakes. Though SEEDE was helpful, he/she may still fail the task within the allotted time (40 minutes).

In the survey, we asked each participant to provide suggestions on how to improve SEEDE to better help software developers in the development/debugging process. 25 of the 29 participants provided suggestions. We found that 19 of the 25 participants thought that the user interface can be improved. In fact, many user interface problems are not really related to SEEDE but to Code Bubbles (e.g., that bubbles of code used in Code Bubbles can be better designed). In terms of the user interface of SEEDE, participants thought that it can be improved to show the values in a more intuitive way and can be simplified. 7 of the 25 participants thought that some parts of SEEDE (e.g., the call tree and the different colors in SEEDE's sliding bar) were confusing. We realized that it is possible that our tutorial might not explain those parts of SEEDE in the best way, although overall the participants think the tutorial was helpful (the average score of usefulness they gave for the tutorial is 4.1). In addition, 2 participants suggested adding new features.

8.3 Threats to Validity

It is challenging to choose the development and the debugging tasks for evaluating SEEDE: they should not be too easy or too hard for the limited time the participants have and should match the programming abilities of the participants. Our selected tasks could be biased in a way to either underestimate or overestimate the effectiveness of SEEDE. The study may also be biased in our usage of college students rather than professional programmers, the varying programming ability of the participants, the relatively small number of users, the relatively small number of tasks, the lack of experience the users had with both Code Bubbles and SEEDE, the limited amount of time spent learning the environment, the time limits imposed on the particular tasks, and possible bugs in both SEEDE and Code Bubbles.

We believe the idea of SEEDE, i.e., providing the run-time executing values in a continuous way, is useful, though the user interface of SEEDE may still be improved.

8.4 Experience in the Wild

SEEDE has been available as an unadvertised and experimental part of Code Bubbles for six months and we have gotten feedback from a limited set of users, primarily in-house.

The first observation is there is a sweet spot for the use of SEEDE. Cases where the fix or the code is small are better done using Java hot swapping because it has lower overhead for a single use. Cases where there are major changes to a large fraction of the system are not suitable for live programming. SEEDE is most useful when the user is working on a bug or writing code that is non-trivial but does not require major changes. Even then, experience showed that it took some time to get used to the facility and to learn the best ways to navigate and use its capabilities.

Otherwise, early experience with the facility has demonstrated its utility. Code Bubbles produces fully anonymous logs for users who opt-in to this facility. Analysis of these logs for the last six months shows that the facility has been used about once every two hours of active use of the environment, with about 100 edits per use. These numbers are probably not representative since usage of the facility includes some experimentation, debugging, and demonstrations.

9 AVAILABILITY

SEEDE is integrated into the currently available Code Bubbles environment. Code Bubbles can be obtained in either open source or binary form at http://www.cs.brown.edu/people/spr/codebubbles. The SEEDE execution engine is available from GitHub (https://github.com/StevenReiss/seede). A demonstration video is available at https://www.youtube.com/watch?v=GpibSxX3Wlw. The detailed results of the study are available upon request.

10 CONCLUSION

Live programming is an interesting concept that can be helpful to the programmer for exploratory programming and debugging. SEEDE demonstrates that it is possible to do live programming for large, complex, object-oriented systems. It provides both an efficient and effective implementation that meets the necessary requirements for practical live programming, and a user interface that lets the programmer see and understand the resultant execution. The practicality of the approach has been demonstrated in a user study.

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 Conference*, pp. 567-580 (1969).

[4] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, Gaurav Seth, and G78-1-4503-4218-6, "Time-travel debugging for JavaScript/Node.js," pp. 1003-1007 in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, , Seattle, WA, USA (2016).

[5] Paolo Boschini, "Live programming for mobile application development," *Uppsala Universitet Master,s Thesis*, (November 2013).

[6] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," *ACM/IEEE International Conference on Software Engineering 2010*, pp. 455-464 (2010).

[7] Dan Bricklin and Bob Frankston, "VisiCalc: information form its creators," *http://danbricklin.com/visicalc.htm*, (1999).

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

[9] E. Bruneton, R. Lenglet, and T. Coupaye, "Asm: a code manipulation tool to implement adaptable systems," *Adaptable and Extensible Component Systems, http://www.objectweb.org/asm/current/asm-eng.pdf*, (November 2002).

[10] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst, "Interactive record/replay for web application debugging," pp. 473-484 in *Proceedings of the 26th annual ACM symposium on User interface software and technology*, , St. Andrews, Scotland, United Kingdom (2013).

[11] Miguel Campusano, Alexandre Bergel, and Johan Fabry, "Does live programming help program comprehension? - A user study with live robot programming," *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools*, (2016).

[12] Paul Chiusano, "Unison: next-generation programming platform," *http://unisonweborg/2015-05-07/about.html*, (2016).

[13] Jonathan Corley, Benedek Izsfficient and Scalable Omniscient Debugging for MDE, and Istversity of Alabama Ph.D. Dissertation, 2016.

[14] Jonathan Corley, Brian P. Eddy, Eugene Syriani, and Jeff Gray, "Efficient and scalable omniscient debugging for model transformations," *Software Quality Journal* ***25***(1) pp. 7-48 (2017).

[15] Robert DeLine and Danyel Fisher, "Supporting exploratory data analysis with live programming," *2015 IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 111-119 (2015).

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

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

[18] Ronald A. Fisher, "On ther interpretation of X2 from contingency tables, and the calculation of P," *Journal of the Royal Statistical Society* **85**(1) pp. 87-94 (1922).

[19] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes, "Mock roles, not objects," pp. 236-246 in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, (2004).

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

[21] Sumit Gulwani, Mikael Mayer, Filip Niksic, and Ruzica Piskac, "StriSynth: synthesis for live programming," *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, (May 2015).

[22] Philip J. Guo, "Online Python tutor: embeddable Web- based program visualization for CS education," *Proceedings SIGCSE 2013*, (2013).

[23] Mouna Hammoudi, Ali Alakeel, Brian Burg, Gigon Bae, and Gregg Rothermel, "Facilitating debugging of web applications through recording reduction," *Empirical Software Engineering*, (2017).

[24] 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).

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

[26] Christoph Hofer, Marcus Denker, and Stephane Ducasse, "Design and implementation of a backward-in-time debugger," *Proceedings of NODe 2006*, pp. 17-32 (2006).

[27] JSFiddle, Ltd., "JSFiddle," *https://jsfiddle.net*, (2017).

[28] Mary Beth Kery, Amber Horvath, and Brad Myers, "Variolite: Supporting Exploratory Programming by Data Scientists," pp. 1265-1276 in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, (2017).

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

[30] Samuel T. King, George W. Dunlap, and Peter M. Chen, "Debugging operating systems with time-traveling virtual machines," pp. 1-1 in *Proceedings of the annual conference on USENIX Annual Technical Conference*, , Anaheim, CA (2005).

[31] 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).

[32] lambdu.org, "Lambu: towards a new programming expereince," *http://www.lambdu.org*, (2017).

[33] Bill Lewis, "Debugging backwards in time," *Proceedings of AADEBUG 2003*, (2003). 225-235

[34] Xiangyu Li, Marcelo d,Amorim, and Alexandro Orso, "Iteraive user-driven fault localization," *Hiafa Verification Conference*, pp. 82-98 Springer International Publishing, (2016).

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

[36] 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).

[37] Tim Mackinnon, Steve Freeman, and Philip Craig, "Endo- testing: unit testing with mock objects," pp. 287-301 in *Extreme programming examined*, Addison-Wesley Longman Publishing Co., Inc. (2001).

[38] Corey Montella, "Eve," *http:://witheve.com*, (2017).

[39] Shaikh Mostafa and Xiaoyin Wang, "Am empirical study on the usage of mocking frameworks in software testing," *14th International Conferernce on Quality Software*, pp. 127-132 (2014).

[40] Nate Murray and Ari Lerner, "Choc: tracable programming," *https://www.fullstack.io/choc/*, (2017).

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

[42] 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).

[43] 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).

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

[45] 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).

[46] Benny Pasternak, Shmuel Tyszberowicz, and Amiram Yehudai, "GenUTest: a unit test and mock aspect generation tool," *Int. J. Softw. Tools Technol. Transf.* ***11***(4) pp. 273-290 Springer-Verlag, (2009).

[47] 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).

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

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

[50] 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).

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

[52] Steven P. Reiss, "Semantics-based code search," *International Conference on Software Engineering 2009*, pp. 243-253 (May 2009).

[53] 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).

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

[55] Manos Renieris and Steven P. Reiss, "ALMOST: exploring program traces," *Proceedings 1999 Workshop on New Paradigms in Information Visualization and Manipulation*, (October 1999).

[56] 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).

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

[58] Walter Savitch, "Absolute Java," *Second edition*, Pearson Addison Wesley, (2003).

[59] Christopher Schuster and Cormac Flanagan, "Live programming for event-based languages," *Proceedings of 2015 Reactive and Event-based Languages and Systems*, (October 2015).

[60] Christopher Schuster and Cormac Flanagan, "Live programming by example: using direct manipulation for live program synthesis," *Proceedings of the LIVE Workshop*, (2016).

[61] Amit Sen, "Liveweave," *http://liveweave.com*, (2017).

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

[63] 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).

[64] Davide Spadini, Maurcio Aniche, Magiel Bruntink, and Alberto Bacchelli, "To mock or not to mock?: an empirical study on mocking practices," pp. 402-412 in *Proceedings of the 14th International Conference on Mining Software Repositories*, (2017).

[65] Josef Stein, *Journal of Computational Physics*1967.

[66] Kevin Su, "Continuous Execution: Improving user feedback in the development cycle," *MIT Sc.M. Dissertation*, (2007).

[67] 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).
[68] Phillip Dale Summers, "Program construction from examples," Yale research report 51 (1976).
[69] Josef Svenningsson, Hans Svensson, Nicholas Smallbone, Thomas Arts, Ulf Norell, and John Hughes, "An Expressive Semantics of Mocking," pp. 385-399 in *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, Springer-Verlag New York, Inc. (2014).
[70] 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).
[71] Bret Victor, "Inventing on Principle," *Talk at CUSEC 2012. Available at https://vimeo.com/36579366*, (2012).
[72] John Vilk, James Mickens, and Mark Marron, "A gray box approach for high-fidelity, high-speed time travel debugging," *https://www.microsoft.com/en-us/research/publication/gray-box-approach-high- fidelity-high-speed-time-travel-debugging/*, (June 2016).
[73] 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).
[74] 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).
[75] Frank Kenneth Zadeck, "Incremental Data Flow Analysis in a Structure Program Editor," *Ph.D. Dissertation, Department of Computer Science, Rice University*, (1984).