Continuous Execution:

Improving user feedback in the development cycle

By

Kevin Su

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 25, 2007

Copyright 2007 Kevin Su. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author

Department of Electrical Engineering and Computer Science

May 25, 2007

Certified by

Rob Miller
Associate Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses
Abstract

Programming environments are increasing the amount of feedback given to users during the development cycle. This work continues with this trend by exploring continuous execution, a programming environment where the result of executing part of a user’s code is outputted as the user writes it. This extra feedback provided to users lets users gain a better understanding of a program’s internals. A prototype implementation is done for Chickenfoot, a JavaScript-based web automation tool. The prototype takes advantage of the web browsing environment of Chickenfoot and uses the browser history to reduce the amount of user code that actually needs to be executed. User testing was done to evaluate the usefulness of continuous execution.
Acknowledgements

I would like to thank Rob Miller for being both my academic advisor as well as my thesis supervisor. His strong guidance has no doubt shaped my entire career at MIT, pushing me towards graduation as well as a bright future. I am extremely thankful to have met Rob and even more grateful to have him as an advisor.

I would like to thank Greg Little, Darris Hupp, Max Goldman, Victoria Chou, and other members of the user interface design group at MIT for their support and advice throughout this process.
Contents

1 Introduction ................................................................................................................................. 7
2 Related Work .............................................................................................................................. 12
  2.1 Barriers in Programming ...................................................................................................... 12
  2.2 Breaking down the understanding and information barriers .............................................. 13
    2.2.1 Debuggers .................................................................................................................... 13
    2.2.2 Reversible Debuggers .................................................................................................. 13
    2.2.3 Testing ......................................................................................................................... 14
  2.3 Improving Feedback at Development Time ......................................................................... 14
    2.3.1 Text editors ................................................................................................................ 14
    2.3.2 Continuous Testing ...................................................................................................... 15
  2.4 Seeing Program Output at Development Time ................................................................. 16
3 User Interface ............................................................................................................................ 17
  3.1 When to execute code .......................................................................................................... 17
  3.2 Executing code ..................................................................................................................... 18
    3.2.1 Multi-line statements .................................................................................................. 20
4 Implementation ........................................................................................................................ 28
  4.1 Detecting line changes ........................................................................................................ 29
  4.2 Extracting code .................................................................................................................... 30
  4.3 Parsing the code .................................................................................................................. 31
  4.4 Filtering code ....................................................................................................................... 34
  4.5 Running code ...................................................................................................................... 35
5 Evaluation .................................................................................................................................. 40
6 Conclusion .................................................................................................................................. 46
  6.1 Future Work ........................................................................................................................ 46
7 References .................................................................................................................................. 48
List of Figures

1.1 The start state of a script. ................................................................. 8
1.2 The line “go('http://sicp.csail.mit.edu/Spring-2007');” is executed. ....... 9
1.3 The script is executed to the end. It inserts the text “INSERT LOG IN FORM
HERE” into the page. .................................................................................. 10
1.4 A script is reverted back to an old state ........................................ 10
1.5 A script is re-executed with an edit made ..................................... 11
3.1 A simple script and its partial execution ........................................ 19
3.2 A simple script with a page load and its partial execution .......... 20
3.3 A for loop illustrating the problem of taking only the code up to the cursor ... 20
3.4 A script is reverted back to an old state ...................................... 21
3.5 A script is re-executed with an edit made ................................... 22
3.6 A script that replaces all the links on the page with the word “link.” .......... 24
3.7 A simple function example ................................................................. 25
4.1 This figure shows the workflow that happens when code is executed .......... 28
4.2 A simple script and its HTML representation .................................. 30
4.3 The HTML of Figure 3.3 in tree form ............................................. 31
4.4 The parse tree of “var x = 3;\nif(x==3) {\noutput(4);\n}” .......................... 32
4.5 A short script and its Interval tree ............................................... 33
5.1 The script used for beginning user evaluation ............................... 40
5.2 Users were shown this window and asked to figure out the script’s behavior. .. 42
5.3 The desired output of user task A .................................................. 43
5.4 The desired output of user task B .................................................. 43
1 Introduction

There are a variety of development environments, all with differing features and capabilities. Many development environments adhere to two popular programming models: the batch model and the interpretive model.

In the batch model, a user writes code, compiles it to obtain an executable, and then runs the executable. An example development environment is one that contains Notepad, a C compiler, and a command prompt. The user writes some C code in Notepad, compiles it from the command line, and then obtains an executable. When the executable is run, if it does not behave as expected, the user edits the code, compiles it again to obtain a new executable, and runs it. This process repeats until the user is satisfied. An inherent flaw with this programming model is the long feedback loop, where it may take a significant amount of time for a programmer to check the behavior of the code that was written.

In the interpretive model, a user can evaluate a line of code as it is written. An example environment would be using Python from the shell in its interactive mode. The user is left with a transcript of what he/she typed, but the transcript itself is not a program.

Development environments have improved upon the feedback given in the batch and interpretive model. They combine the advantages of the batch programming model and the interpretive programming model, allowing a user to see what effect a line of code has on a program as it is written, and still end up with some executable script. Other development environments provide tools to give more information than just the output to users and some even provide templates for users to write code.

For example, Eclipse, a Java development environment, uses a compiler and a parser during the writing of code, and gives users continuous feedback about syntactic errors and type errors. Eclipse provides a debugger which allows users to incrementally move through a program, seeing the state of the program after each line is executed. And it also provides unit testing capabilities, to improve the user’s ability to thoroughly test their code. These tools and many more have improved user feedback and provided valuable assistance during the development cycle.
This work continues the trend of improving user feedback during the development cycle. It does this by creating a continuous execution environment, which executes the user’s code and outputs the results of that code as it is being written. Specifically, this work creates a continuously executing environment for Chickenfoot.

Chickenfoot is a Mozilla Firefox browser plug-in that allows users to alter their web experiences without having to view or manipulate the underlying HTML source. It does this by providing a development environment that contains all the functionality of JavaScript with additional wrapper functions to facilitate web-page navigation and manipulation. By making code continuously execute in this context, users can immediately see the results of how scripts affect their web experiences.

In Figure 1.1, we see a step-by-step example of Chickenfoot enhanced with continuous execution being used to develop a short Chickenfoot script and its effect on the HTML (output) window.

![Figure 1.1 The start state of a script.](image-url)
Figure 1.2 The line “go(http://sicp.csail.mit.edu/Spring-2007);” is executed when the ‘Enter’ button is hit.

The goal of continuous execution is to help the user acquire information about code as it is written. To do this, continuous execution outputs the result of executing all the code up to but not including the cursor line as the user edits. Users can see intermediate steps of their program, to see if it is behaving as expected.

Figures 1.1, 1.2, 1.3, 1.4, and 1.5 show the evolution of the HTML window as a script is developed. In Figure 1.3, we see a script executed to completion. In Figure 1.4, the HTML window has reverted back to the stage where only “go('sicp.csail.mit.edu/Spring-2007')” has been executed, to reflect that only the code up to the cursor line was run. And finally, in Figure 1.5, the code is again completely executed, with the results of the edit showing in the HTML window. We can see that by continuously executing the code, users are presented with the ability to see the state of the script after a particular line is executed.
Figure 1.3 The script is executed to the end. It inserts the text “INSERT LOG IN FORM HERE” into the page.

Figure 1.4 Line 2 of the script is edited. Notice that the script has returned to the same state as it was in Figure 1.2.
This extra feedback has a few benefits. It provides:

- **Help with syntax errors:** Users can see the execution of a line of code immediately as they navigate the script, with error messages appearing in the Chickenfoot console if there are syntax errors. A user can see an error happen as soon as a particular line is edited and navigated away from, allowing them to infer what the error was and what line caused it, correct it, and learn from it.

- **Help with behavioral errors:** Similarly to syntax errors, users are shown the result of their code and are able to compare it to the desired effect.

- **Help with comprehending code:** Seeing what a particular line of code does to a program allows the user to gain a better understanding of what that line contributes to the program.

The rest of this thesis discusses details of the design and implementation of a continuous execution environment set in the Chickenfoot framework. Chapter 2 discusses related work. Chapter 3 will discuss implementation details. Chapter 4 talks about a preliminary evaluation of continuous execution in Chickenfoot, and Chapter 5 details what still needs to be done and where to go from here.
2 Related Work

Much work has considered how to make the development cycle easier for users. First, we try to understand the barriers that exist in developing code, and how we can overcome those barriers. After understanding those barriers, we will examine a number of tools and concepts that try to overcome those barriers and give users more feedback and assistance as they develop code.

2.1 Barriers in Programming

Ko and Myers [5] identified six different kinds of barriers associated with programming. Design barriers deal with cognitive difficulties of a programming problem. Selection barriers are caused by properties of an environment’s facilities for finding what programming interfaces are available and which can be used to achieve a particular behavior. Coordination barriers are a programming system’s limits on how interfaces in its language can be combined to achieve complex behaviors. Use barriers are properties of a programming interface that obscure how to use it. Understanding barriers are properties that obscure what a program does when it runs. Finally, information barriers make it difficult to acquire information about a program’s internals.

Continuous execution attempts to breakdown the understanding barrier. An environment with continuous execution allows users to see the results of a program as it is written, helping to clarify a program’s behavior when it is run.

Another barrier that continuous execution tackles is the information barrier. With continuous execution, users can see the program state as it’s executed up to a particular line, and then navigate downward to see what that line does. This lends insight into what that particular line is doing, and thus allows the user to acquire information about a program’s internals.
2.2 Breaking down the understanding and information barriers

Many existing tools help break the understanding and information barriers. These tools give users the extra capabilities beyond just execution to figure out what their code is doing.

2.2.1 Debuggers

Debuggers are one common option available to users that attack the information barrier. After writing some amount of code, users can use a debugger to help them step through their code in an attempt to understand it. To use the debugger, a user first sets a series of breakpoints and enters a debugging mode. In the debug mode, the user can run the code up to selected breakpoints, at which point the user can start stepping incrementally through the code. These steps can either be as small as one line of code, or as large as a function, depending on the user’s needs. At each point in the debug mode, a list of variables and their values is available to the user. Using this variable list, the user can acquire information about the program’s current state, and making sure that the behavior of the program is as it should be.

2.2.2 Reversible Debuggers

An improvement on debuggers is the concept of reversible debuggers. According to Delisle, Menicosy, and Schwartz [3], debugging accounts for about half of a programmer’s total time. One problem with current debuggers is the inability to move backwards through code. According to Koju, Takada, and Doi [7], software does not fail immediately after the cause of a particular program failure is executed. That is, the line where an exception is thrown, or where the program stops working as expected, is not usually the line that contains the error.

Reversible debuggers remedy the problem by also offering the ability to step backward, allowing execution to the time of failure and then backtracking to where it originated.
2.2.3 Testing

Users can generate tests for their code to gain feedback about it. For example, Java has JUnit. JUnit provides a framework for users to test cases for modules and their methods, using assert statements to determine if a given module or method is behaving as expected by running it on a number of examples. These examples are generated manually by the user, who has to think of common cases as well as corner cases. Alternatively, a user can write scripts to automatically generate test cases.

All of these options provide users help with breaking down the information and understanding barrier. However, they are not used at the same time code is written. In order to develop code, a user has two phases: a writing phase where code is written and a testing phase where the user can use these tools. Continuous execution merges these two phases. It mimics the capabilities of a reversible debugger, where a user can step backwards or forwards from some point in a script, while constantly seeing the program output.

2.3 Improving Feedback at Development Time

Debuggers, reversible debuggers, and unit testers allow users to gather valuable feedback from their code, but must do this independently from writing code. There are currently numerous tools that actually provide users with feedback about their code during the writing process.

2.3.1 Text editors

Structured text editors [11] gave users code snippets and templates to assist in writing code, containing features like a content assist list, which contained functions defined by the syntax of the language in which the user was coding, and could be used to auto-complete blocks of code. Users could also define their own code snippets for regularly used bits of code. It was believed that code snippets and templates would cut down on development time, as users could select templates rather than write long blocks of code. This would prevent them from making mistakes on templated code. They also provided syntax highlighting and unlimited undo/redo.
Structured text editors eventually lost out to free-form text editors which simply allowed users to directly manipulate what they were writing. Early models of freeform text editors provided users with some minimal feedback at development time. Some provided auto-indenting and syntax coloring using simple regular expressions, to give users some minimal feedback about the correctness of what they are writing.

Newer free-form text editors have increased the feedback given to users. For example, Eclipse, an integrated development environment which includes a free-form text editor, parses code and compiles it to give users feedback about syntactic errors as well as type errors. The parser is run constantly while the compiler is run every time a file is saved. Eclipse is just one example of a number of integrated development environments that give users more and more feedback at development time.

2.3.2 Continuous Testing

A step further in providing users with feedback as code is written is continuous testing [5]. Continuous testing uses excess cycles on a developer’s workstation to continuously run tests in the background to provide feedback to users about failures as they edit code. The point of it is to reduce the effort required to keep code well-tested, and thus well understood, and to prevent errors from being uncaught for long periods of time, helping to catch bugs. An experimental evaluation of continuous testing during development showed a significant effect on success of completing a programming task, while having little to no effect on the amount of time worked.

An application of continuous testing was Example-Centric Programming [3]. Example-Centric Programming presented a prototype of an IDE that tried to turn the REPL (Read-Eval-Print Loop) into a single unified tool.

The read-eval-print loop (REPL) is a simple programming environment. It consists of 3 main functions: read, eval, and print. The read function accepts an expression from the user, and parses it into some data structure in memory. The eval function takes that data structure and evaluates it. The print function takes the result from eval, and prints it out to the user.

Example-centric programming merges those three functions. The idea behind example-centric programming is to have a user create some simple test cases (maybe
even regression tests) for a particular program they are writing. Then while they are writing code, the examples are tested, and the user can see the evolution of their examples and how it approaches the solution during its development. This behavior helps users identify a program’s behavior and help break down the understanding and information barriers.

This idea of letting users see the results of their code on test cases is taken in another direction with continuous execution. Instead of just doing test cases, continuous execution in Chickenfoot shows the output of part of a script as it is written, letting the user know the internal behavior of the script immediately. This feedback during code development helps users overcome the understanding barrier and the information barrier.

2.4 Seeing Program Output at Development Time

Finally, there are programs where users can see the results of their actions almost immediately. For example, one of the most commonly used is Microsoft Excel spreadsheets. Spreadsheets allow users to enter lines of code into cells (often simple branching logic), and show the output of that code in the cell immediately as it is entered. The output of the program is the spreadsheet itself and is constantly updated as the user edits it.

There are also a number of graphical user interface (GUI) languages that have continuously executing development environments. One example is XML Markup Language (XUL), which has a development environment called Extension Developer, written for Mozilla Firefox [12]. Extension Developer comes with a XUL editor, which has two windows: one for editing code and one for displaying the output of the code. As a user edits code, its result is displayed in the output. This clearly helps greatly with catching errors, as any incorrectly formed statement causes the output window to display an error message.

This work combines the advantages of the discussed tools to further improve user feedback. It has the capabilities of a reversible debugger to step through code and showing its output, and it does this as the user writes code. This work gives users a tool that integrates writing code and testing in an elegant and integrated fashion.
3 User Interface

Continuous execution outputs the results of executing a user’s code as it is written. It is a read-eval-print loop where the user can hit return to obtain some feedback about the code and still end up with some finished program. It is also a debugger where breakpoints can be set and inspected. The difference for continuous execution is that setting breakpoints and inspecting them is automatic based on where the user puts the cursor. Users can inspect the code at any point in the program by simply position the cursor.

This extra feedback and the functionality to obtain it should not interfere with user workflow. As such, enhancing Chickenfoot with a continuous execution environment adds no extra user interface elements to the editor. That way, if a user chooses not to use continuous execution at all, then the user’s development process remains unhindered. There is still a risk of distraction caused by the continuous execution environment, and this is addressed in the evaluation section.

3.1 When to execute code

Continuous execution should be designed so that as a user writes code, the effect of each edited line on the output is immediately apparent. This can be achieved by executing the user’s code constantly. This amount of time between executions should be small enough so the output appears to be continuously updating. In this way, the output window would consistently reflect the user’s code, and the user would know exactly what their code is doing. However, this approach is not feasible. Continuous execution needs to run the user’s code, which in the case of Chickenfoot can be a series of network transactions that may be expensive to run repeatedly given the limits of server and network latency.

A more realistic approach is to execute the user’s code on every key stroke and mouse press that occurs in the Chickenfoot editor’s buffers. This is more reasonable because we are listening for discrete key and mouse presses to execute as opposed to trying to simulate the continuity of time. Users can see how each character they write
affects their overall script. However, even this reduced number of executions can become too processor intensive, with more complicated scripts.

Continuous execution needed a way to further reduce the number of executions of the program and still mimic constant output. The solution this work uses is to execute the user's code each time the user changes lines in the Chickenfoot editor. This approach scales back from being able to see individual character changes, to being able to see how individual lines of code affect the output. This approach still faces a similar problem of server and network latency limitations given complicated scripts, but scales a little better than the previous suggestions. However, it is still capable of providing users with constant feedback about the output of their code.

3.2 Executing code

There are a few decisions to be made when deciding what code to run. One option is to run all the code in the buffer. If the entire buffer is evaluated, users can see the output of the program as a whole, but lose the ability to see the effect of individual lines. A second choice is to evaluate a single line; specifically the line the cursor is on. A persistent environment can be maintained to remember the evaluation of lines. If just the current line is evaluated, users could see what any particular line did by navigating through the script, but the functionality of multiple lines of code integrated together becomes obscure.

The approach taken by this work is a compromise between these two choices that closely resembles the breakpoint model of debuggers. The cursor position is automatically set as a breakpoint, and code is run up to that breakpoint. For example, in Figure 3.1, the output window displays the results of lines 1-7, but not 8, 9, or 10. Because we are executing on line-change events, a user can see the partial execution of the script, and figure out if the program is behaving correctly at intermediate steps by navigating through the lines. A typical use case can be to navigate between 2 lines, comparing the changes made to the output and determining the behavior of the lines.
The behavior is different when the user’s script contains page loads. As shown in Figure 3.2, the line “go (‘google.com’)” causes a page load. The cursor is on the line “output (8)”. Only the code between these two lines is executed. This behavior is a side effect of the way code is executed. To save on expensive page loads, this implementation of continuous execution uses the browser history to jump to different points in the script. This is detailed in the implementation section. Skipping parts of the script causes the effect seen in Figure 3.2, where only some of the JavaScript is executed.
3.2.1 Multi-line statements

The behavior we want is for all the code up to but not including the cursor line to be executed. If every statement was a single line, this would work just fine. However, there are statements that span multiple lines, such as control-flow statements. When the cursor is positioned in the middle of a multi-line statement, a couple problems occur.

Figure 3.3 A for loop illustrating the problem of taking only the code up to the cursor.
One problem is that syntax errors are presented to the user, even for correctly formed statements. As seen in Figure 3.3, the code up to but not including the cursor is:
for (word = new find('Google'); word.hasMatch; word = word.next) {

When this code is executed, a missing '}' syntax error is displayed in the output console, despite the fact that what is written is all syntactically correct. This problem is easily fixed by adding closing brackets to the executed code.

A more serious problem occurs with control-flow statements, where a single line of code can potentially be executed multiple times like in a for loop, or not executed at all like in an else-if statement. It is not obvious what to execute when the cursor is positioned inside one of these statements. This work establishes a number of heuristics to deal with control-flow statements.

For multi-line conditional statements, if the cursor is located within an if statement’s body, the code up to the cursor line is chosen for execution with closing brackets added to the end to make it syntactically correct. From Figure 3.5, the extracted code would be:

1 if (x == 1) {
2 output('one');
3 }
4 else if (x==2) {
5 output('two');
6 }
7 else {
8 output('three');
9 }

The cursor lies within the else-if multi-line statement, which has a depth of 1, and therefore one closing bracket is added.

---

Figure 3.4 A multi-line branching example.
An alternative approach for conditional statements is to execute only the code that is enclosed by the conditional statement as the cursor. In Figure 3.4, this would mean that regardless of the value of \( x \), we execute \texttt{output('two')}. Although this behavior benefits users by letting them see what each branch of their 	exttt{else-if} constructs did, it also ignores variable values which obscures the true program state.

There are some JavaScript branching statements that are not supported in the prototype system, like 	exttt{switch} statements and 	exttt{try/catch} blocks. However, because these statements are so similar in structure and behavior to 	exttt{if-else} constructs, it is likely that the same heuristics will be used.

For multi-line loop statements such as 	exttt{for} and 	exttt{while}, the heuristic this work uses is the same as that of an \texttt{if} statement. All the code up to the cursor line is extracted, and closing brackets are added. An example of the behavior can be seen in Figure 3.5.

![Figure 3.5a](image)

\textbf{Figure 3.5a} The code up to line 2 is executed, with the for loop currently doing nothing.
Figure 3.5b  All the code before the cursor is executed, i.e. all but one line of the for loop.

Looking at Figure 3.5, we see that the lines of code up to the cursor line are executed. For Figure 3.5a, the code before the for loop and a for loop with no body is executed. For 3.5b, the line up to the ‘replace’ command is looped over and for 3.5c,
the script finally fully executes. As the user navigates through a loop, he/she can see how each line changes the loop, and thus changes the overall program.

This approach of executing the iterations of loops in parallel is a bad idea if each iteration of the loop depends on the state of the iterations before it. As shown in Figure 3.6, we see such an example. This loop terminates if \texttt{find("link")} doesn't return a match. However, the second line is responsible for replacing all possible matches on the page with the word "link" and if it is not executed, then \texttt{find("link")} always returns a match, resulting in an infinite loop.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.6a.png}
\caption{A script that replaces all the links on the page with the word "link."}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.6b.png}
\caption{The same script that doesn't work because of the way lines are chosen to be executed.}
\end{figure}
A fix is to prevent loops from executing infinitely by injecting the following code into all loops:

```plaintext
1 if (n > MAX_LOOP_ITERATIONS) {
2     break;
3 }
4 n++;
```

The variable n is initialized to 0 before the loop begins and MAX_LOOP_ITERATIONS is some number of our choosing. This prevents loops like these from looping infinitely. It should be large enough to allow the user to see the effect of the for loop, but small enough to not spend too much time in a wasted loop.

An alternative approach to executing all the code up to the cursor line is to set the cursor line as the breakpoint. The user decides to see the first n iterations of a loop. The code inside the loop is completely executed n times. On the n+1\textsuperscript{st} iteration, the code is only executed up to the breakpoint. This lets the user potentially see every step of the loop by varying the cursor position and the number of iterations. However, this approach requires adding an extra user interface elements to allow users to select the number of iterations. One of the goals of this work is to enhance user feedback without interfering with the user’s workflow, and thus this approach was abandoned.

For functions, if the cursor lies inside a function body, then all the code up to the cursor line is appended with all the code after the end of the function body and is executed.

![Chickenfoot Script Ecktor0](image)

**Figure 3.7** A simple function example.
The code that gets executed in Figure 3.7:

```javascript
1 function test() {
2   output(1);
3   output(2);
4 }
5 output('Begin test');
6 test();
```

The script is run to the end, ignoring any code inside the function that appears on or after the cursor line. This is drastically different than the behavior for branching statements and loops, which simply stopped at the cursor line. This is because functions are themselves very different from other types of block statements. The body of a function can be executed multiple times at different sections of the program, depending on where it is called. Other block statements are positioned in one section of the code. In Figure 3.7, lines 1 through 4 only define a function. If we remove line 6, the function is never executed. Therefore, we have to treat functions differently from other multi-line statements, as executing code up to the cursor line doesn’t make sense for functions.

An alternative approach to executing the script to its end when inside a function body is to use the cursor line as a breakpoint. Only the code up to the breakpoint is executed and the program stops there. The code that is actually outputted from Figure 3.7 is then:

```javascript
1 output('Begin test');
2 output(1);
3 output(2);
```

This is because the code before the function call is executed, and the code inside the function is executed up to the cursor. The execution would terminate here, meaning later calls to `test()` would not be executed.

The following script presents a problem for the first approach.

```javascript
1 function test() {
2   var x = 3;
3   return x;
4 }
5 output(test());
6 output(test());
```

For this example, if the cursor was on line 3, the first approach would meaninglessly output nothing, because the return statement was never reached. The
second approach would stop at line 2, which is consistent with the behavior had the code been written in-line instead of called from a function.

However, the first approach does have its advantages over the second one.

```javascript
1 function test(num) {
2     output(2*num);
3     output(4*num);
4 }
5 test(17);
6 test(23);
7 test(46);
```

For this example, if the cursor was on line 3, the second approach would stop after only the first call to test, outputting only 34. The second approach would allow the user to see the execution of the function on multiple inputs rather than just the first.

The first approach was selected because of its ability to let a user debug all the function calls rather than just the first one. And though its behavior is different, it still meets the goal of providing feedback to users. Users moving the cursor inside a function body can see how changes made inside of functions affect the program.
4 Implementation

In order to create the effect of continuously executing and outputting code, the continuous execution environment for Chickenfoot goes through a number of steps. The first steps decide when to execute code and figure out what code to execute. That code is then executed and its result is displayed. This workflow is graphically illustrated in Figure 4.1.

Figure 4.1 shows the four modules that are each responsible for a step in executing the user’s code: Extraction, Parser, Filter, and Evaluation. The extraction module is responsible for starting the process. It listens for the appropriate events in the

![Diagram of the workflow](image.png)

**Figure 4.1** This figure shows the workflow that happens when code is executed.
Chickenfoot editor’s current buffer, extracts all of the buffer’s code, and then passes it to the parser. The parser takes the extracted code, parses it to figure out relevant information, and forwards the code and the parsed information to the filter module. The filter module eliminates lines of code according to the parsed information. The filtered code is passed to the evaluation module, which evaluates the code and displays the result through the available output channels, which in Chickenfoot are the HTML window and the Chickenfoot output console.

4.1 Detecting line changes

Line changes are easily detectable. The method `getSelection()` provided by XUL which returns an `NSISelection` object. The `NSISelection` object represents the selected text in the editor and has the following properties:

- `anchorNode` – the node that the selection begins in
- `anchorOffset` – the offset from the beginning of the node that the selection starts at.

The cursor is just selected text, where the size of the selected text is 0. Using `anchorNode`, a method called `getCursorLine()` gets the current cursor line by depth first searching through the HTML tree representing the buffer code, counting the number of `<BR>` nodes and stopping when the `anchorNode` was reached. After each key press, if `getCursorLine()` is different than the last cursor line, a line-change event has occurred.

A problem with executing on every single line change is that if a user wants to change lines very quickly, then the continuous execution environment will still try to execute many times in a short period.

The solution to this was to further reduce the number of executions is to not run on every single line-change event. Instead, we only execute after a line-change event if no other line-change events occurred within the last 250 milliseconds.
4.2 Extracting code

The Chickenfoot editor is an HTML editor so its display is represented by HTML. As shown in Figure 4.2, the HTML for the Chickenfoot editor is responsible for laying out the buffer and keeping track of new lines and white space. This work takes advantage of the HTML tree structure in order to iterate over the nodes and extract the code.

Figure 4.3 is the HTML that we are working with as a tree. Performing a depth first search on the tree results in passing over the buffer lines, and thus the text, in the correct order (from top to bottom and left to right). So to extract the code from the buffer, we use a wrapped version of the `createTreeWalker()` method provided by the DOM to perform a depth first search. In our depth first search, for every text node we encounter we append its `nodeValue` to a temporary string variable. For every element node, we check to see if it is a `<br>` element, and append an appropriate "n" to our string.
The end result is a string for this particular example is the following string:
"var x = 3; \nif(x==3) {\noutput(4); \n}"

This string is then passed on to the parsing step.

4.3 Parsing the code

On a high level, the parsing module takes the code from the extraction module, parses it with Rhino, and returns a tree of information later used in filtering the code. Rhino[13] is an open-source implementation of JavaScript written in Java. Rhino contains a JavaScript parser which is used for this work. An instance of ErrorLogger, an object that keeps track of errors in the parsed JavaScript, is needed to instantiate Rhino's parser. When Rhino parses code, this ErrorLogger is populated with any problems Rhino had while parsing the code. Thus, figuring out the errors is a matter of accessing the list of errors from the ErrorLogger.

This work uses the Rhino parser to find the beginnings and endings of statements that span multiple lines, such as loops and conditionals, the significance of which was discussed in the user interface chapter and illustrated in Figure 3.3.
Figure 4.4 The parse tree of "var x = 3; \nif (x==3) { \noutput (4); \n}\n"

When the extracted code is passed to the parser, it returns a tree of Nodes. Each Node has a type, describing it as a function node, a branch node, a variable node, etc. Each Node also contains pointers to its children, which are Nodes themselves. For example, if we take our extracted string example "var x = 3; \nif (x==3) { \noutput (4); \n}\n" and have Rhino parse it, we get the tree seen in Figure 4.4.

This parse tree returned by Rhino is used to determine the start and end lines of a multi-line statement. Because multi-line statements can be nested within each other, it is natural to represent them as another tree. An Interval class was created to represent multi-line statements. Each Interval class has the following data members:

- startLine – the start line
- endLine – the end line
- parent – the enclosing multi-line statement
- children – the multi-line statements inside of this one
- type – the type of node, e.g. loop
- depth – how deep the Interval is in the tree

Using the parse tree of nodes returned by Rhino, we create a tree of Intervals. A depth-first search is performed on the parse tree. At each step, we keep track of the current depth as well as a stack of Intervals we are currently inside of (the stack structure handles nesting of multi-line statements). For each Node, determine the line it starts on and the line it ends on using the getLineNo() method of Node. Whenever we encounter a Node that spans multiple lines, construct an Interval to represent that Node and make it a child of the top of the Interval stack, i.e. the Interval that we are currently inside of. The current Interval is then pushed onto the stack, signifying that we are inside that Interval.

```java
for(var i = 0; i<100; i++) {
    if(i%2==0){
        if(i%4==0) {
            System.out.println("Multiple of 4");
        }
        else {
            System.out.println("Even");
        }
    } else {
        System.out.println("Odd");
    }
}
```

**Figure 4.5** A short script and its Interval tree. The numbers in the nodes are “start : end.”
When we find a Node of depth less than the node on top of the stack, we know that we have exited the Interval and the stack is popped. We continue like this until the depth first search is completed. The end result is a tree structure representing our nested Intervals within the code.

4.4 Filtering code

With the information provided by the parsing module, the code can be filtered according to the heuristics described in the user interface section. This means that we only execute code up to the cursor line. For if statements and loops we automatically fix syntax problems with curly braces, and for functions this means running the entire script, with only the part of the function up to the cursor line run. For example:

```java
if (true) {
    output(1);
    if (true) {
        output(2);
        output(3);
    }
    output(4);
}
```

If the cursor was on line 5, the code that would be executed is:

```java
if (true) {
    output(1);
    if (true) {
        output(2);
    }
}
```

These heuristics were easy to implement for branching statements and loop statements and any nesting of the two. The algorithm was:

1. Find the current cursor line.
2. Find the deepest nested Interval that the cursor lies within.
3. Get all the code up to the cursor line and add curly braces based on the depth of the Interval to fix syntax problems.

However, functions presented a minor issue. The parse tree that Rhino returns unfortunately did not parse anything inside of a function unless it was explicitly called somewhere in the executed code. When it did parse, the line numbers were relative to
where the function was called, and not relative to where the function was defined. This made it impossible figure out multi-line statements that were nested inside of a function.

The solution to this was to treat the function body like a miniature buffer and extract the code from within the function body, pass it to Rhino and create a tree of the Intervals inside the function. This tree of Intervals can then be used to determine what code inside the function should be run based on our defined heuristics. This filtered code, instead of going to an output window is passed upwards as the code to be run for the function body.

4.5 Running code

Now we need to execute the filtered code. The trivial solution is to run all of the filtered code. This is what current continuous execution environments like Firefox Extension Developer do for XUL. However, the reason why this isn’t viable for continuous execution in Chickenfoot is because while Extension Developer executes XUL code, Chickenfoot scripts are essentially JavaScript code, which can be significantly more complicated and take much longer to execute due to the potential network transactions performed by the script. For example:

```
1    go('google.com')
2    click('advanced')
3    output('hi');
```

This code blocks until google.com is loaded and then the advanced search page is loaded. Depending on the network connection speed, this can take a significant amount of time. Page loads are the most expensive operation in the scripts, but all of the pages are recorded in the browser history. This work exploits the browser history by using it to jump to some intermediate point of the script that caused a page load, and then executes only the portion of the code not covered by the jump.

To access the browser history, this work accesses the web navigation object provided by the Firefox development environment. The web navigation object provides simple functions like goToIndex(int i) which browses to the ith link from the beginning the browser history (so the 0th link is the page that has been in the history the longest) and get(int i) which gets the link at index i.
This work creates a mapping between the script to be run and the browser history, where each line that causes a page load is given a number \( n \) such that \( \text{get}(n) \) returns the link that corresponds to that page load. This mapping is updated as the user’s code is executed.

To create the initial mapping, the beginning of the script is marked as a checkpoint and it is mapped to the initial browser state. This is so that the script can revert to its start state when the user moves the cursor to the first. The code to be executed is then run one line at a time. As this code is run, we use a \texttt{SHistoryListener} XUL object accessible through the Firefox chrome window to listen for changes made to the history. This history listener is capable of detecting when new pages are added to the history, or when the forward and backward buttons are pressed. Each line that adds a new page to the history is marked as a checkpoint. We do this until the script has been fully executed. Then we iterate through the lines marked as checkpoints in order excluding the initial checkpoint added, numbering them from 1 to \( n \), where \( n \) is the number of new pages added to the browser history. We can verify what point in history each checkpoint corresponds to by simply using \( \text{get}(i) \). By our construction, there are an equal number of checkpoints as there are pages added to the history. In this case, the code is run completely without taking advantage of the browser history at all.

When the mapping is not empty, we have an opportunity to shortcut some code. We first determine the latest point in the mapping that is valid, or any point that is still consistent with the script. Specifically, any time an edit is made in the script editor since the last mapping was updated, its line number is recorded in a variable called \texttt{earliest_edit_index}. Any checkpoints that occur after the \texttt{earliest_edit_index} are invalidated and removed from the mapping. After jumping to the latest point in the mapping, we proceed as in the empty case; by executing the code that still needs to be executed line by line, listening for new pages being added to the browser history, and marking checkpoints.

There are a couple of problems with this approach. The first problem is if checkpoints occur inside of a loop. One line would correspond to multiple lines in
browser history. The current prototype implementation does not handle checkpoints inside of a loop.

The second problem is keeping the mapping consistent with the cursor position. If the user decides to manually navigate through the browser history, then the output HTML window will no longer reflect the correct position in the script.

One possible solution for this approach is to listen for forward or backward events (possible with SHistoryListener) and move the cursor in the editor for the user. This option is still being explored.

The used solution to fixing this inconsistency is to delete the mapping and recreate it, thus making the mapping consistent with the script again. This occurs after n executions of the user’s code. Currently, this is implemented by setting the earliest_edit_index to be 0, essentially saying that every point in the mapping is invalid and needs to be recreated.

However, this approach creates a different problem with maintaining a correct JavaScript evaluation context. Let’s examine the following example:

```javascript
1  go('google.com');
2  var googleList = '<ul>";
3  for (m = find("link"); m.hasMatch; m=m.next) {
4      googleList += '<li>"+m.element;
5  }
6  go('web.mit.edu/ksulwd/www/blank.html');
7  insert(after('Google'), googleList);
8

Assuming the mapping is not empty, when the user navigates to line 8, the code up to line 6 is shortcutted by loading from the browser history, and then only line 7 is executed. However, googleList at line 7 is not defined and the script won’t work properly. All the statements are correctly formed and variables appropriately defined, so why does this problem happen?

The problem lies in the code that we are shortcutting. If we had run the code normally, the variable would be created and put into the JavaScript evaluation context. However, we instead tried to shortcut some of the code by using the browser history. The browser history only contains the pages that the user has been through and doesn’t contain any extra state not saved on those pages. In this case, the variable googleList was
part of the JavaScript state, and thus by simply loading a page from browser history, its
value was not loaded correctly.

One solution is for each checkpoint to also store the correct JavaScript evaluation
context for that point in the script. Then each time we shortcut some code, we load the
appropriate page from browser history, as well as the appropriate evaluation context,
correctly loading any JavaScript variables defined at that point in the script.

However, maintaining a copy of a context for each individual checkpoint was
problematic. We can easily get fresh JavaScript evaluation contexts by exposing the
iFrame behind the Chickenfoot editor and grabbing the JavaScript evaluation context
from it. However, keeping these contexts consistent with their checkpoint is difficult.
For example, if we just shortcutted to some checkpoint and loaded the appropriate
evaluation context, we need to evaluate the rest of the script with respect to that context,
but the context itself should remain unchanged. That is, we don’t want to mutate our
evaluation context to be inconsistent with its checkpoint, so we can reuse it in the future
if we return to this checkpoint. However, evaluating any code with respect to a particular
evaluation context risks potentially changing that context as well.

One suggested solution was to load a context, deep copy it, and then evaluate
code with respect to the copy. However, deep copying in JavaScript is also difficult, as
there are cyclic properties (properties that link to each other) contained within each
evaluation context.

The implemented solution this work uses is a persistent evaluation context, where
all the code that has been executed is remembered. If we create a variable and assign it a
value, it always has that value until we specifically reassign it. This was implemented by
creating a variable in each buffer of the Chickenfoot editor that stores the current
evaluation context. All code in a particular buffer is evaluated with respect to the global
context variable of the buffer. Thus any variables made in a script for a particular buffer
are always defined within that buffer’s context, though the variables may not necessarily
contain the correct value.

A second problem caused by shortcutting code is related to the JavaScript state in
forms. The browser remembers changes made to a form and upon loading the page,
those changes are also loaded. For example, if we ran this script:
Line 2 is a checkpoint, as it causes the advanced image search page to be added to the history. When we navigate down to line 4, the word ‘hi’ is be entered into the form, and ‘GIF’ is selected in one of the dropdown menus. When we navigate back up to line 2, the state is returned to when the advanced image search page was first browsed to, i.e. a fresh advanced search page. However, what shows up is the advanced image search page with ‘hi’ entered and ‘GIF’ selected. This is a problem because the effect of lines that change forms becomes invisible. For example, because ‘hi’ is already entered, navigating from line 3 to line 4 would have no visible effect on the output.

A proposed solution was to create undo commands for each Chickenfoot command. Whenever a line is navigated upwards from, we execute its undo command to reverse its effect.

Having the ability to undo commands would also expand the ways we could shortcut code. Instead of shortcutting to some checkpoint within the current script and executing the code after the checkpoint, we could shortcut to a check point that occurs after the code we want to run and step backwards via undo commands to the cursor line. Using a combination of the two different ways of shortcutting code could further reduce the amount of code that gets executed. However, undo commands are still being researched and not currently implemented.
5 Evaluation

The continuous execution environment for Chickenfoot underwent a few preliminary usability tests, to see if the users understood that their script was continuously being executed and its result was constantly being outputted. It was also of interest to see if users could take advantage of the extra information provided to them.

The first user test was aimed towards beginning programmers. These users were given a script of simple Chickenfoot commands and asked to identify the behavior of each of these commands. Users were told that they could run the script either by clicking the Chickenfoot run button, or by moving the cursor through the script using the up and down arrows. The script was:

```
1    go('images.google.com');
2    click('advanced');
3    enter('all words', 'hi');
4    pick('gif');
5    pick('small');
6    click('google search');
7    for (m = find('www'); m.hasMatch; m=m.next) {
8        replace(m, 'WWW result');
9    }
```

Figure 5.1(a) Users were shown this window when the cursor was at line 3.
Figure 5.1(b) Users were shown this window when the cursor was at line 6.

Figure 5.1(a) Users were shown this window when the cursor was at line 10.

Figure 5.1 shows examples of what users saw during the evaluation as they navigated upwards and downwards through the script. Three users were tested in this way, all of which had no programming experience. In all cases, the users were able to identify the correct behavior for lines 1, 2, 6 and 8. However, at least one user was unable to identify the behavior of lines 3, 4, 5, 7, or 9. Every user used the up and down arrows and moved the cursor through the script. After each test, users were asked what helped them identify the lines that they did. One user claimed that it was obvious from the line what should happen. “It makes sense that go('images.google.com'); would bring you to images.google.com” or “I saw pick('gif') and I saw a box with the word ‘GIF’ in it and put it together”. The other two users had similar guesses for line behavior, but
added that their guesses were confirmed upon seeing the page change as they moved the cursor passed the line.

The lines that some users failed to identify did not change the page. Examples of lines that did not change the page are 7 and 4. Line 7 didn’t change the page because it was simply a for loop. Line 4 is a problem with browser side effects. The browser itself stores the state of a form and any changes made to it. In this case, the change of selecting the GIF option in a dropdown menu was saved, causing the line “pick(‘gif’)” to have no effect. This suggests that users found the changing of the output windows useful in confirming the behavior of a line and advocates that browser side effects created by commands like “pick(‘gif’)” need to be addressed.

The second user test was conducted using experienced programmers, each having at least 3 years in the computer science program at MIT, and each having at least 6 months of JavaScript programming experience. Users in this test were given two tasks.

**Task A:** In this task, users were placed in front of a computer with the figure in 5.2 presented to them. They were also given a picture of the web page in Figure 5.3. Users were first asked to modify the given script to convert the web page in Figure 5.2 to the web page in Figure 5.3.

![Figure 5.2](image)

**Figure 5.2** Users were shown this window and asked to figure out the script’s behavior.
Figure 5.3 The desired output of task A.

Task B: Users were asked to write code that gets all the links the Google home page and inserts those links in an unordered list after the word ‘Google’ on the page web.mit.edu/ksulwd/www/blank.html, as shown in Figure 5.4. Users were expected to remember the code from task A in order to accomplish task B.

Figure 5.4 The desired output of task B.
For both tasks, users were also given a reference sheet, listing functions they might encounter and details about how to use them. During these two tasks, the users were observed, paying special attention to moments where users struggled. After users completed the tasks, they were asked if they understood what they environment was doing, and whether they found it useful.

Of the four users tested, all completed the tasks given to them. All users were able to figure out that the code up to their cursor was constantly being executed and outputted. When asked if they found the output useful, one user claimed that it was nice to receive confirmation that a line worked as they thought. A second user said that it was kind of like having print outs of the line they were working on. In general, the users felt this continuous execution environment would be helpful in script development, although one worried that it would be very distracting.

Users were noticeably affected by the continuous execution environment. All four users spent some time navigating the script for task A before they even started coding. One user stopped completely for 20 seconds when the output window was first changed. This suggests that the users had trouble figuring out what the environment was doing or that they were just distracted by its constant changing.

For task B, one user began to write a for loop, completed it correctly, hit return, and was presented with a syntax error, reporting a missing ‘}’. He then spent some time checking over his for loop, making sure it was correct, and then asking if there was anything wrong with it. This brings to light a problem with continuous execution in how to deal with half completed multi-line statements.

Though this work automatically adds closing brackets to fix syntax issues, it only does this on correctly formed statements. In this example, a partially-formed for loop produced a syntax error. The reason why this work does not complete partially-formed statements is because if a partially-formed loop was executed, the user can misinterpret that to mean that it is correctly written.

A second possibility is to continue to output errors for malformed statements, but change the error messages to be more detailed. In the case of the user who had trouble with task B, an error message that says “Partially formed loop” would have been more informative than “Syntax error.” A third possibility is to automatically close partially
formed multi-line statements automatically. Eclipse does something like this, where a closing parenthesis/bracket is automatically paired with an opening parenthesis/bracket.
6 Conclusion

Continuous execution improves user feedback in the development cycle. In Chickenfoot, it shows users an HTML window and an output console that details the results of script execution as it is being developed. It does this without adding any extra UI elements. The informal evaluation suggests first that we need to refine the feedback given to users by continuous execution and second that we need to perform more user tests to discover issues with continuous execution.

6.1 Future Work

One way to improve feedback is in addition to showing the codes output at intermediate steps, to also show the variable state as debuggers do. Currently, if the user creates a variable with a line like “var x = 3”, no part of the output is changed. A variable state window would remedy this, showing the evolution of the variables’ values as the script is executed.

A second way to improve feedback is showing the users more clearly what code is being run. The user tests suggest that the behavior of the continuous execution environment was not intuitive. Perhaps executing code up to but not including the cursor line was a bit obscure. One way to disambiguate is to actually show the user what is run and what is not. This could be accomplished by perhaps graying out the code that is not run slightly, or drawing a line between the code that is run and is not run.

Finally, it is plausible that one set of users will think executing code up to but not including the cursor line is intuitive, while another set of users think that executing the cursor line as well is more intuitive. However, we can remove the guess work that users have to do by creating two output windows, one containing the output of all the code up to but not including the cursor line, and one containing the output of all the code up to and including the cursor line. In our current model, the user has to navigate to a line, and alternate hitting the down and up arrow to see the before and after states. With this approach, the user could see the effect of a line by simply moving the cursor to it, and
observing the difference between the two output windows. This adds an extra user interface element, but greatly increases the user's feedback.

There is still a lot of research and work that can be done for continuous execution environments. Creating a continuous execution environment for Chickenfoot is a first step in that research. More user testing to increase the amount of useful feedback continuous execution environments provide and decreasing the unnecessary feedback is another potential next step.

Finally, as the continuous execution environment for Chickenfoot is developed, extending into other mainstream languages like Java, and C++ is a possibility. In this way, we can hope to continue improving user feedback for all languages in all integrated development environments.
7 References


