An Experimental Evaluation of Continuous Testing During Development

David Saff Michael D. Ernst

MIT Computer Science & Artificial Intelligence Lab
The Stata Center, 32 Vassar Street
Cambridge, MA 02139 USA
{saff,mernst}@csail.mit.edu

ABSTRACT

Continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background, providing rapid feedback about test failures as source code is edited. It is intended to reduce the time and energy required to keep code well-tested and prevent regression errors from persisting uncaught for long periods of time. This paper reports on a controlled human experiment to evaluate whether students using continuous testing are more successful in completing programming assignments. We also summarize users' subjective impressions and discuss why the results may generalize.

The experiment indicates that the tool has a statistically significant effect on success in completing a programming task, but no such effect on time worked. Participants using continuous testing were three times as likely to complete the task before the deadline. Participants using continuous compilation were twice as likely to complete the task, providing empirical support to a common feature in modern development environments. Most participants found continuous testing to be useful and believed that it helped them write better code faster, and 90% would recommend the tool to others. The participants did not find the tool distracting, and intuitively developed ways of incorporating the feedback into their workflow.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: [Integrated environments]; D.2.5 [**Testing and Debugging**]: [Testing tools]

General Terms

Experimentation, Human Factors, Measurement, Verification

Keywords

continuous testing, unit testing, continuous compilation, test-first development

1. INTRODUCTION

Continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background as the developer edits code. It provides developers rapid feedback regarding errors that they have inadvertently introduced. Continuous testing is inspired by continuous compilation, a feature of many modern development environments that gives rapid feedback about compilation errors. This paper experimentally evaluates whether the extra feedback from continuous testing assists developers in a programming task without producing harmful side effects.

It is good practice to use a regression test suite while performing development tasks such as enhancing or refactoring an existing codebase. (Test-driven development [3] seeks to extend this situation to all development tasks: before each piece of functionality is added, a test for the functionality is written, added to the suite, and observed to fail.) During development, running the test suite bolsters the developer's confidence that they are making steady progress, and catches regression errors early. The longer a regression error persists without being caught, the larger its drain on productivity: when the error is found, more code changes must be considered to find the changes that directly pertain to the error, the code changes are no longer fresh in the developer's mind, and new code written in the meanwhile may also need to be changed as part of the bug fix.

Running tests has a cost: remembering to run the tests, waiting for them to complete, and returning to the task at hand distract from development. In order to avoid running the tests too often or too seldom, developers often wait until they are least confident in the code. Unfortunately, some of the hardest errors to fix are introduced by what the developer believes to be an innocuous change. Not only is the developer unlikely to run the tests soon after such a change, but they are unlikely to remember the change as a possible source of the error when they do finally discover the problem.

Developers may employ test case selection [16, 25] and prioritization [36, 26] to reduce the cost of running tests. They may also continue editing the code while tests run on an old version, but this further complicates reproducing and tracking down errors.

Continuous testing uses real-time integration with the development environment to asynchronously run tests against the current version of the code and notify the developer of regression errors. The version of the code being tested is constantly kept in sync with the version being edited. Whenever the thinking time between two edits is long enough to run at least one test, continuous testing can provide useful feedback, without requiring any attention from the developer unless a regression error is found. Continuous testing can be combined with selection, prioritization, or other approaches for further optimization. The developer no longer has to consider when to run the tests, and errors are caught more quickly, especially those that the developer had no cause to suspect.

Previous work prospectively evaluated continuous testing [28]. Developer behavior was monitored at a fine granularity, including the state of all editor buffers and all on-disk files, and when tests were run. These observations permitted determination of the *ignorance time* between introduction of each regression error (in the developer's editor) and the developer becoming aware of the error (by running the test suite), and the *fix time* between the developer

ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.

becoming aware of the error and fixing the error. The ignorance time and fix time are related: larger ignorance times yield larger fix times. This suggests that reducing ignorance time should reduce fix time; this observation, along with the development history, permit prediction of how much time could have been saved by use of a tool that reduced ignorance time (and indirectly reduced fix time). The previous work suggested that continuous testing could have reduced development time by 10–15% for two single-developer software projects, which is substantially more than would have been achieved by changing manual test frequency or reordering tests.

This paper reports a study that evaluated whether the extra feedback provided by continuous testing improved the productivity of student developers, without producing detrimental side effects like distraction and annoyance. In a controlled human experiment, 22 students in a software engineering course completed two unrelated programming assignments as part of their normal coursework. The assignments supplied partial program implementations and tests for the missing functionality, simulating a test-first development methodology. All participants were given a standard Emacs development environment, and were randomly assigned to groups that were provided with continuous testing, just continuous compilation, or no additional tools. Data was gathered from remote monitoring of development progress, questionnaires distributed after the study, and course records. In all, 444 person-hours of active programming were observed.

The experimental results indicate that students using continuous testing were statistically significantly more likely to complete the assignment by the deadline, compared to students with the standard development environment. Continuous compilation also statistically significantly increased success rate, though by a smaller margin; we believe this is the first empirical validation of continuous compilation. Our analysis did not show a higher success rate for students who frequently tested manually. Furthermore, the tools' feedback did not prove distracting or detrimental: tool users suffered no significant penalty to time worked, and a large majority of study participants had positive impressions of the tool and were interested in using it after the study concluded.

This paper is organized as follows. We first detail the tools provided to students (Section 2) and the design of the experiment (Section 3). We then present quantitative (Section 4) and qualitative (Section 5) results and threats to validity (Section 6). Finally, we discuss related (Section 7) and future work (Section 8) and conclude with a summary of findings (Section 9).

2. TOOLS

We have implemented a continuous testing infrastructure for the Java JUnit [13] testing framework and for the Eclipse [8] and Emacs [9] development environments.

JUnit is a regression testing framework for Java. It supports assertions for checking expected results, organizing test cases into hierarchical suites, running a suite, and presenting results textually or graphically. Our JUnit extension (used by both plug-ins) persistently records whether a test has ever succeeded in the past. This permits it to both change the order in which tests are run and the order in which results are printed. For instance, regression errors, which are more likely to be surprising to the developer, can be prioritized over unimplemented tests.

We have built continuous testing plug-ins for both Eclipse [30] and Emacs. We focus here on the Emacs plug-in, which was used in our experiment. We describe how the user is notified of problems in his or her code and how the plug-in decides when to run tests. We conclude this section with a comparison to pre-existing features in Eclipse and Emacs.

Because Emacs does not have a standard notification mechanism, we indicated compilation and test errors in the mode line. The mode line is the last line of each Emacs text window; it typically indicates the name of the underlying buffer, whether the buffer has unsaved modifications, and what Emacs modes are in use. The Emacs plug-in (a "minor mode" in Emacs parlance) uses some of the empty space in the mode line. When there are no errors to report, that space remains blank, but when there are errors, then the mode line contains text such as "Compile-errors" or "Regressions:3". This text indicates the following states: the code cannot be compiled; regression errors have been introduced (tests that used to give correct answers no longer do); some tests are unimplemented (the tests have never completed correctly). Because space in the mode line is at a premium, no further details (beyond the number of failing tests) are provided, but the user can click on the mode line notification in order to see details about each error. Clicking shows the errors in a separate window and places the cursor on the line corresponding to the failed assertion or thrown exception. Additional clicks navigate to lines corresponding to additional errors and/or to different stack frames within a backtrace.

The Emacs plug-in performs testing whenever there is a sufficiently long pause[1]; it does not require the user to save the code, nor does it save the code for the user. The Emacs plug-in indicates errors that would occur if the developer were to save all modified Emacs buffers, compile, and test the on-disk version of the code. In other words, the Emacs plug-in indicates problems with the developer's current view of the code.

The Emacs plug-in implements testing of modified buffers by transparently saving them to a separate shadow directory that contains a copy of the software under development, then performing compilation and testing in the shadow directory. Users never view the shadow directory, only the test results. This approach has the advantage of providing earlier notification of problems. Otherwise, the developer would have no possibility of learning of problems until the next save, which might not occur for a long period. Notification of inconsistent intermediate states can be positive if it reinforces that the developer has made an intended change, or negative if it distracts the developer; see Section 5.

The continuous testing tool represents an incremental advance over existing technology in Emacs and Eclipse. By default, Emacs indirectly indicates syntactic problems in code via its automatic indentation, fontification (coloring of different syntactic entities in different colors), indication of matching parentheses, and similar mechanisms. Eclipse provides more feedback during editing (though less than a full compiler can), automatically compiles when the user saves a buffer, indicates compilation problems both in the text editor window and in the task list, and provides an integrated interface for running a JUnit test suite. Our Emacs plug-in provides complete compilation feedback in real time and JUnit integration, and both of our plug-ins provide asynchronous notification of test errors.

3. EXPERIMENTAL DESIGN

This section describes the experimental questions, subjects, tasks, and treatments in our evaluation of continuous testing.

3.1 Experimental questions

Continuous testing exploits a regression test suite to provide more feedback to developers than they would get by testing manually, while also reducing the overhead of manual testing. Continuous

[1]Tests are run after 5 seconds of idle time, if it has been at least 15 seconds or 30 keystrokes since the last test run.

testing has intuitive appeal to many developers, but others are skeptical about its benefits. An experimental evaluation is required to begin to settle such disagreements.

We designed an experiment to address these issues, in hopes of gaining insight into three main questions.

1. Does continuous testing improve developer productivity? Increased developer productivity could be reflected either by accomplishing a task more quickly, or by accomplishing more in a given timeframe. We could control neither time worked nor whether students finished their assignments, but we measured both quantities via monitoring logs and class records.

2. Does the asynchronous feedback provided by continuous testing distract and annoy users? Intuition is strongly divided: some people to whom we have explained the idea of continuous testing have confidently asserted that continuous testing would distract developers so much that it would actually make them slower. To address this question, we used both qualitative results from a participant questionnaire and quantitative productivity data as noted above.

3. If the continuous testing group was more productive, why? Continuous testing subsumes continuous compilation, and it enforces frequent testing; perhaps one of those incidental features of the continuous testing environment, or some other factor, was the true cause of any improvement.

   (a) Continuous compilation. We compared the performance of three treatment groups: one with continuous testing, one with only continuous compilation, and one with no additional tool.

   (b) Frequent testing. Although all students were encouraged to test throughout development, not all of them did so. We compared the performance of students who tested more frequently with those who tested less frequently and with those who had a continuous testing tool. Any effect of easy or fast testing was unimportant, since all students could run the tests in five seconds with a single keypress.

   (c) Demographics. We chose a control group randomly from the participants, and we measured and statistically tested various demographic factors such as programming experience.

3.2 Participants

Our experimental subjects were students, primarily college sophomores, in MIT's 6.170 Laboratory in Software Engineering course (`http://www.mit.edu/~6.170`). This is the second programming course at MIT, and the first one that uses Java (the first programming course uses Scheme). Of the 100 students taking the class during the Fall 2003 semester, 34 volunteered to participate in the experiment. In order to avoid biasing our sample, participants were not rewarded in any way. In order to maintain consistency between the experimental and control groups, we excluded all volunteers who did not use the provided development environment for all of their development. The excluded students used a home PC or a different development environment for some of their work. This left us with 22 participants for the first task, and 17 for the second task (see Section 3.3).

On average, the participants had 3 years of programming experience, and one third of them were already familiar with the notions of test cases and regression errors. Figure 1 gives demographic details regarding the study participants.

	Mean	Dev.	Min.	Max.
Years programming	2.8	2.9	0.5	14.0
Years Java programming	0.4	0.5	0.0	2.0
Years using Emacs	1.3	1.2	0.0	5.0
Years using a Java IDE	0.2	0.3	0.0	1.0

	Frequencies
Usual environment	Unix 29%; Win 38%; both 33%
Regression testing	familiar 33%; not familiar 67%
Used Emacs to compile	at least once 62%; never 38%
Used Emacs for Java	at least once 17%; never 83%

Figure 1: Study participant demographics (N=22). "Dev" is standard deviation.

Don't use Emacs	45%
Don't use Athena	29%
Didn't want the hassle	60%
Feared work would be hindered	44%
Privacy concerns	7%

Figure 2: Reasons for non-participation in the study (N=31). Students could give as many reasons as they liked.

Differences between participants and non-participants do not affect the internal validity of our study, because we chose a control group (that was supplied with no experimental tool) from among the participants who had volunteered for the study. Our analysis indicates that it would have been wrong to use non-participants as the control group, because there were statistically significant differences between participants and non-participants with respect to programming experience and programming environment preference.

Two factors that we measured predicted participation to a statistically significant degree. First, students who had more Java experience were less likely to participate: participants had an average of .4 years of Java experience, whereas non-participants had an average of almost .8 years of Java experience. Many of the more experienced students said this was because they already had work habits and tool preferences regarding Java coding. Overall programming experience was not a predictor of participation. Second, students who had experience compiling programs using Emacs were more likely to participate; this variety of Emacs experience did not predict any of the factors that we measured, however.

Figure 2 summarizes the reasons given by students who chose not to participate; 31 of the non-participants answered a questionnaire regarding their decision. Very few students indicated that privacy concerns were a factor in their decision not to participate, which was encouraging considering the degree of monitoring performed on the students (see Section 3.5). The 6.170 course staff only supported use of Athena, MIT's campus-wide computing environment, and the Emacs editor. In the experiment, we provided an Emacs plug-in that worked on Athena, so students who used a different development environment could not participate. The four non-Emacs development environments cited by students were (in order of popularity): Eclipse, text editors (grouping together vi, pico, and EditPlus2), Sun ONE Studio, and JBuilder. Students who did not complete their assignments on Athena typically used their home computers. Neither student experience with, nor use of, Emacs or of Athena was a statistically significant predictor of any measure of success (see Section 4.2).

3.3 Tasks

During the experiment, the participants completed the first two assignments (problem sets) for the course. Participants were treated

	PS1	PS2
participants	22	17
skeleton lines of code	732	669
written lines of code	150	135
written classes	4	2
written methods	18	31
time worked (hours)	9.4	13.2

Figure 3: Properties of student solutions to problem sets. All data, except number of participants, are means. Students received skeleton files with Javadoc and method signatures for all classes to be implemented. Students then added about 150 lines of new code to complete the programs. Files that students were provided but did not need to modify are omitted from the table.

	PS1	PS2
tests	49	82
initial failing tests	45	46
lines of code	3299	1444
running time (secs)	3	2
compilation time (secs)	1.4	1.4

Figure 4: Properties of provided test suites. "Initial failing tests" indicates how many of the tests are not passed by the staff-provided skeleton code. Times were measured on a typical X-Windows-enabled dialup Athena server under a typical load 36 hours before problem set deadline.

no differently than other students. All students were encouraged by staff and in the assignment hand-out to run tests often throughout development. The problem sets were not changed in any way to accommodate the experiment, nor did we ask participants to change their behavior when solving the problem sets. All students were given a 20-minute tutorial on the experimental tools and had access to webpages explaining their use. A few students in the treatment groups chose to ignore the tools and thus gained no benefit from them.

Each problem set provided students with a partial implementation of a simple program. Students were also provided with a complete test suite (see Section 3.3.1). The partial implementation included full implementations of several classes and skeleton implementations of the classes remaining to be implemented. The skeletons included all Javadoc comments and method signatures, with the body of each method containing only a RuntimeException. No documentation tasks were required of students for these assignments. The code compiled and the tests ran from the time the students received the problem sets. Initially, most of the tests (all those that exercised any code that students were intended to write) failed with a RuntimeException; however, some tests initially passed, and only failed if the student introduced a regression into the provided code.

The first problem set (PS1) required implementing four Java classes to complete a poker game. The second problem set (PS2) required implementing two Java classes to complete a graphing polynomial calculator. Both problem sets also involved written questions, but we ignore those questions for the purposes of our experiment. Figure 3 gives statistics regarding the participant solutions to the problem sets.

3.3.1 *Test suites*

Students were provided with JUnit test suites prepared by the course staff (see Figure 4). Passing these test suites correctly accounted for 75% of the grade for the programming problems in the problem set. Each test case in the suites consists of one or more method calls into the code base, with results checked against expected values.

The suites are optimized for grading, not performance, coverage, or usability. (That is, the test cases were developed and organized according to the pedagogical goals of the class.) However, experience from teaching assistants and students suggests that the tests are quite effective at covering the specification students were required to meet. Compiling and testing required less than five seconds even on a loaded dialup server, since the suites were relatively small (see Figure 4). Thus, there was no point in using test prioritization or selection when running these particular test suites.

It is rare in professional development for a developer to develop or receive a complete test suite for the desired functionality before they begin writing code on a new development project. However, since the students were encouraged to concentrate on one failing test at a time, the effect of the development scenario was similar to the increasingly common practice of test-driven development [3]. The task also had similarities to maintenance, where a programmer must ensure that all tests in a regression test suite continue to succeed. Finally, when developers are striving for compatibility or interoperability with an existing system, a de facto test suite is available, since the two systems' behavior can be compared.

Several deficiencies of the provided test suites and code impacted their usefulness as teaching tools and students' development effectiveness. The PS1 test suite made extensive use of test fixtures (essentially, global variables that are initialized in a special manner), which had not been covered in lecture, and were confusing to follow even for some experienced students. In PS2, the provided implementation of polynomial division depended on the students' implementation of polynomial addition to maintain several representation invariants. Failure to do so resulted in a failure of the division test, but not the addition test. Despite these problems, students reported enjoying the use of the test suites, and found examining them helpful in developing their solutions. Figure 5 gives more detail about student use of the provided test suites, ignoring for now participants who used continuous testing; note that students who volunteered for the study (even if their data was later excluded), and thus knew they were being monitored, were more likely to report testing throughout development.

	volunteers	non-volunteers
waited until end to test	31%	51%
tested throughout	69%	49%
test frequency (minutes)		
mean	20	18
min	7	3
max	40	60

Figure 5: Student use of test suites, self-reported. "Volunteers" omits those who used continuous testing, but includes students who volunteered for the study, but were excluded for using an IDE other than Emacs. Only students who tested regularly throughout development reported test frequencies.

3.4 Experimental treatments

The experiment used three experimental treatments: a control group, a continuous compilation group, and a continuous testing group. The control group was provided with an Emacs environment in which Java programs could be compiled with a single keystroke and in which the (staff-provided) tests could be run with a single keystroke. The continuous compilation group was additionally provided with asynchronous notification of compilation errors in their code. The continuous testing group was further provided with asynchronous notification of test errors. The tools are described in Section 2.

For each problem set, participants were randomly assigned to one of the experimental treatments: 25% to the control group, 25% to the continuous compilation group, and 50% to the continuous testing group (a larger group, to increase opportunities for qualitative feedback from users). Thus, most participants were assigned to different treatments for the two problem sets; this avoids conflating subjects with treatments and also permits users to compare multiple treatments.

3.5 Monitoring

Participants agreed to have an additional Emacs plug-in installed on their system that monitored their behavior and securely transmitted logs to a central remote server. The logged events included downloading the problem set, remotely turning in the problem set, changes made to buffers in Emacs containing problem set source (even if changes were not saved), changes to the source in the file system outside Emacs, and clicking the mode line to see errors. A total of 444 person-hours, or almost 3 person-months, of active time worked were monitored. These logs afforded us additional predictor and criterion variables beyond those of Section 4.2. They did not yield statistically significant results, so we omit them from this paper for brevity, but full details may be found elsewhere [27].

4. QUANTITATIVE RESULTS

This section reports on quantitative results of the experiment; Section 5 gives qualitative results.

4.1 Statistical tests

This paper reports all, and only, effects that are statistically significant at the $p = .05$ level. All of our statistical tests properly account for mismatched group and sample sizes.

When comparing nominal (also known as classification or categorical) variables, we used the Chi-Square test, except that we used Fisher's exact test (a more conservative test) when 20% or more of the the cells of the classification table had expected counts less than 5, because Chi-Square is not valid in such circumstances. When using nominal variables to predict numeric variables, we used factorial analysis of variance (ANOVA). Where appropriate, we determined how many of the effects differed using a Bonferroni correction.

When using numeric variables as predictors, we first dummy-coded or effect coded the numeric variables to make them nominal, then used the appropriate test listed above. We did so because we were less interested in whether there was a correlation (which we could have obtained from standard, multiple, or logistic regression) than whether the effect of the predictor on the criterion variable was statistically significant in our experiment.

4.2 Variables compared

We used 20 variables as predictors: experimental treatment, problem set, and all quantities of Figures 1 and 8.

The key criterion (effect) variables for success are:

- time worked. Because there is more work in development than typing code, we divided wall clock time into 5-minute intervals, and counted 5 minutes for each interval in which the student made any edits to the `.java` files comprising his or her solution.
- errors. Number of tests that the student submission failed.
- correct. True if the student solution passed all tests.
- grade, as assigned by TAs. We count only points assigned for code; 75% of these points were assigned automatically based on the number of passed test cases.

Treatment	N	Correct
No tool	11	27%
Continuous compilation	10	50%
Continuous testing	18	78%

Figure 6: Treatment predicts correctness. "N" is the number of participants in each group. "Correct" means that the participant's completed program passed the provided test suite.

4.3 Statistical results

Overall, we found few statistically significant effects. We expected this result, because most effects are not statistically significant or are overwhelmed by other effects — either ones that we measured or other factors such as the well-known large variation among individual programmers. (Another reason might be the relatively small sample size, but the fact that some effects were significant suggests that the sample was large enough to expose the most important effects.) This section lists all the statistically significant effects.

1. Treatment predicts correctness (see Figure 6). This is the central finding of our experiment, and is supported at the $p < .03$ level. Students who were provided with a continuous testing tool were three times as likely to complete the assignment correctly as those who were provided with no tool. Furthermore, provision of continuous compilation doubled the success rate. The latter finding highlights the benefit that developers may already get from the continuous compilation capabilities included in modern IDE's such as Eclipse [8] and IDEA [12].
2. Continuous testing vs. regular manual testing predicts correctness. Students were asked in the online questionnaire whether they tested throughout development (see Figure 5). Of participants who were not given continuous testing, but reported testing throughout, only 33% successfully completed the assignment, significantly less than the 78% success rate for continuous testing. There was no statistically significant difference in test frequency between complete and incomplete assignments within this group. The mean frequency for manual testing (see Figure 5) among those who tested throughout was once every 20 minutes, which is longer than the mean time to pass an additional test during development (15 minutes), possibly indicating that students were often writing code to pass several tests at a time before running the tests to confirm.
3. Problem set predicts time worked (PS1 took 9.4 hours of programming time on average, compared to 13.2 hours for PS2). Therefore, we re-ran all analyses considering the problem sets separately. We also re-ran all analyses considering only successful users (those who submitted correct programs).
    (a) For PS1 only, years of Java experience predicted correctness and grade. For the first problem set, participants with previous Java experience had an advantage: 83% success rate and average grade 74/75, versus 14% success rate and average grade 61/75 for those who had never used Java before. By one week later, the others had caught up (or at least were no longer statistically significantly behind).

Treatment	N	time worked		errors		grade	
No tool	11	10.1	(3.7)	7.6	(8.2)	59	(21)
Cont. comp.	10	10.6	(1.6)	4.1	(6.9)	62	(9)
Cont. testing	18	10.7	(4.3)	2.9	(6.9)	64	(11)

Figure 7: Effect of treatment on other success variables, as defined in Section 4.2. "N" is the number of participants in each group. "Time worked" is in hours for successfully completed tasks only. "Errors" and "grade" are for all participants. Means are shown followed by standard deviations in parentheses. No effect was statistically significant at the $p = .05$ level.

(b) For PS1 participants with correct programs, years of Java IDE experience predicts time worked: those who had previously used a Java IDE spent 7 hours, compared to 13 hours for the two who reported never previously using a Java IDE. No similar effect was observed for any other group, including PS1 participants with incorrect programs or any group of PS2 participants.

It is worth emphasizing that we found no other statistically significant effects. In particular, of the predictors of Section 4.2 (including user perceptions of the experimental tools), none predicted number of errors, time worked, or grade, except for the effects from experience seen in PS1. The only effects on student performance throughout the study period were the effects that continuous testing and continuous compilation had in helping significantly more students complete the assignment.

Figure 7 shows the effects, which were not significant at the $p = .05$ level, of treatment for the success variables other than completion. Time worked (among participants who completed the assignment) shows no trend, which was counter-intuitive to us; based on previous suggestive results [28], we had expected continuous testing to reduce time worked. It may be that continuous compilation and continuous testing truly had no affect on time worked, or it may be that true positive or negative effects were masked by Parkinson's Law: "Work expands to fill the time available for its completion." That is, students may have budgeted a certain amount of time to the problem set, worked toward that budget by either rushing to complete if they were behind, or gold-plating for code clarity if they were ahead, and turned in whatever they had when time ran out.

5. QUALITATIVE RESULTS

We gathered qualitative feedback about the tools from three main sources. All students were asked to complete an online questionnaire containing multiple-choice and free-form questions. We interviewed staff members about their experiences using the tools, helping students with them, and teaching Java while the tools were running. Finally, some students provided direct feedback via e-mail.

Section 5.1 discusses the results of the multiple choice questions. The remainder of this section summarizes feedback about changes in work habits, positive and negative impressions, and suggestions for improvement.

5.1 Multiple choice results

Figure 8 summarizes the multiple-choice questions about experiences with the tools. Participants appear to have felt that continuous compilation provided somewhat more incremental benefit than continuous testing (though the statistical evidence of Section 4.3 indicates the opposite). Impressions about both tools were positive overall.

	Continuous compilation (N=20)	Continuous testing (N=13)
The reported errors often surprised me	1.0	0.7
I discovered problems more quickly	2.0	0.9
I completed the assignment faster	1.5	0.6
I wrote better code	0.9	0.7
I was distracted by the tool	-0.5	-0.6
I enjoyed using the tool	1.5	0.6
The tool changed the way I worked	1.7	
I would use the tool in 6.170	yes 94%; no 6%	
… in my own programming	yes 80%; no 20%	
I would recommend the tool to others	yes 90%; no 10%	

Figure 8: Questionnaire answers regarding user perceptions of the continuous testing tool. The first 6 questions were answered on a 7-point scale ranging from "strongly agree" (here reported as 3) through "neutral" (reported as 0) to "strongly disagree" (reported as -3). The behavior change question is on a scale of 0 ("no") to 3 ("definitely").

The negative response on "I was distracted by the tool" is a positive indicator for the tools. In fact, 70% of continuous testing and continuous compilation participants reported leaving the continuous testing window open as they edited and tests were run. This confirms that these participants did not find it distracting, because they could easily have reduced distraction and reclaimed screen space by closing it (and re-opening it on demand when errors were indicated).

Especially important to us was that 94% of students wanted to continue using the tool in the class after the study, and 80% wanted to apply it to programming tasks outside the class. 90% would recommend the tool to others. This showed that developers enjoyed continuous testing, and most did not have negative impressions of distraction or annoyance.

5.2 Changes in work habits

Participants reported that their working habits changed when using the tool. Several participants reported similar habits to one who "got a small part of my code working before moving on to the next section, rather than trying to debug everything at the end." Another said, "It was easier to see my errors when they were only with one method at a time." The course staff had recommended that all students use the single-keystroke testing macro, which should have provided the same benefits. However, some participants felt they only got these benefits when even this small step was automated.

This blessing could also be a curse, however, exacerbating faults in the test suites (see Section 3.3.1): "The constant testing made me look for a quick fix rather than examine the code to see what was at the heart of the problem. Seeing the 'success' message made me think too often that I'd finished a section of code, when in fact, there may be additional errors the test cases don't catch."

5.3 Positive feedback

Participants who enjoyed using the tools noted the tools' ease of use and the quickness with which they felt they could code. One enjoyed watching unimplemented tests disappear as each was correctly addressed. Several mentioned enjoying freedom from the mechanical tedium of frequent manual testing: "Once I finally figured out how it worked, I got even lazier and never manually ran the test cases myself anymore." One said that it is "especially useful for someone extremely prone to stupid typo-style errors, the kind that are obvious and easily fixable when you see the error line but which don't jump out at a glance."

Staff feedback was predominantly positive. The head TA reported, "the continuous testing worked well for students. Students used the output constantly, and they also seemed to have a great handle on the overall environment." Staff reported that participants who were provided the tools for the first problem set and not for the second problem set missed the additional functionality.

Several participants pointed out that the first two problem sets were a special case that made continuous testing especially useful. Full test suites were provided by the course staff before the students began coding, and passing the test suite was a major component of students' grades on the assignments. Several participants mentioned that they were unsure they would use continuous testing without a provided test suite, because they were uncomfortable writing their own testing code, or believed that they were incapable of doing so. One said that "In my own programming, there are seldom easily tested individual parts of the code." It appears that the study made participants think much more about testing and modular design, which are both important parts of the goals of the class, and are often ignored by novice programmers. The tools are likely to become even more useful to students as they learn these concepts.

5.4 Negative feedback

Participants who didn't enjoy using the tools often said that it interfered with their established working habits. One said "Since I had already been writing extensive Java code for a year using emacs and an xterm, it simply got in the way of my work instead of helping me. I suppose that, if I did not already have a set way of doing my coding, continuous testing could have been more useful." Many felt that the reporting of compilation errors (as implemented in the tool) was not helpful, because far too often they knew about the errors that were reported. Others appear to have not understood the documentation. Several didn't understand how to get more information by clicking on the errors reported in the modeline.

Some participants believed that when the tool reported a compilation or test error, that the tool had saved and compiled their code. In fact, the tool was reporting what would happen were the user to save, compile, and test the code. Some users were surprised when running the tests (without saving and compiling their work) gave different results than the hypothetical ones provided by the tool.

5.5 Suggestions for improvement

Participants had many suggestions for improving the tools. One recommended more flexibility in its configuration. (As provided, the tools were hardcoded to report feedback based on the staff-provided test suite. After the study completed, students were given instructions on using the tools with their own test suite.) Another wanted even more sophisticated feedback, including a "guess" of why the error is occurring. Section 9 proposes integrating continuous testing with Delta Debugging [37], to provide such a hint.

5.5.1 *Implementation issues*

Some students were confused because continuous testing tool filtered out some information from the JUnit output before displaying it. In particular, it removed Java stack frames related to the JUnit infrastructure. These were never relevant to the code errors, but some users were alarmed by the differences between the continuous testing output and the output of the tests when run directly with JUnit.

When a test caused an infinite loop in the code under test, no continuous testing feedback appeared. This is identical to the behavior of standard JUnit, but since students had not manually run the tests, some thought that the tool had failed.

Some participants reported an irreproducible error in which the results appeared not to change to reflect the state of the code under particular circumstances. One participant reported that this happened 2 or 3 times during the two weeks of the study. These participants still reported that they would continue using the tools in the future, so we assume it was not a huge impediment to their work.

The most common complaint and improvement recommendation was that on compute-bound workstations (such as a 333-MHz Pentium II running a modern operating system and development environment, or a dialup workstation shared with up to 100 other users all running X applications), the background compilation and testing processes could monopolize the processor, sometimes so much that "cursor movement and typing were erratic and uncontrollable." One said that "it needs a faster computer to be worthwhile." However, most students found the performance acceptable. We conclude that potential users should be warned to use a system with acceptable performance, and that additional performance optimizations are worthwhile.

6. THREATS TO VALIDITY

Our experiment has produced statistically significant results showing that for student developers using a test-first methodology, a continuous compilation tool doubles the success rate in creating a correct program, and a continuous testing tool triples the success rate. However, the circumstances of the experiment must be carefully considered before applying the results to a new situation.

One potential problem with the experiment is the relative inexperience of the participants. They had on average 2.8 years of programming experience, but only 0.4 years of experience with Java. Two thirds of them were not initially familiar with the notion of regression testing. More experienced programmers might not need the tools as much — or they might be less confused by them and make more effective use of them. Student feedback suggests that the benefits are most available to users who are open to new development tools and methodologies, not those who are set in their ways (see Sections 3.2 and 5.4).

The participants were all students; this is both a strength and a weakness. On the plus side, use of students has several benefits. The subject demographics are relatively homogeneous, each subject performed the same task, the subjects were externally motivated to perform the task (it was part of their regular coursework, not an invention of the experimenters), and most subjects were exposed to two experimental treatments. It would be very difficult to reproduce these conditions with professional developers [1]. For this reason, controlled experiments in software engineering commonly use students.[2] On the minus side, use of students limits our ability to generalize these results to other software developers. However, the results strongly suggest that continuous compilation and continuous testing are valuable at least for beginning programmers. The enthusiastic adoption by professionals of programming environments offering continuous compilation suggests that its benefits are not limited to students. The qualitative feedback from students, from TAs, and from other experienced programmers leads us to believe that the benefits of continuous testing, too, will apply more broadly. This hypothesis has been partially confirmed by a very positive user response from industrial developers to the public beta release of our continuous testing plug-in for Eclipse [30]. We emphasize, however, that there is not yet quantitative evidence of general benefit. Were such evidence desired, it

[2]We reviewed the last two proceedings of each of ICSE, FSE, and ISSTA. Of 5 papers that reported new controlled experiments on humans [4, 7, 18, 19, 38], all used students.

would require a series of experiments with a broad range of programmers and tasks. Such experiments would complement the one we have performed, in that they would be likely to have lesser external threats to validity but greater internal threats to validity.

The experiment involved relatively few participants. We addressed the risk of type I error (a false alarm) by only reporting statistically significant effects; the statistical tests account for the number of data points. There is also a risk of type II error (a failed alarm): only relatively large differences in effects are statistically significant. There may be other effects that are less important (smaller in magnitude) yet would become statistically significant, given a larger data set. The statistics that are available to us at least indicate the most important effects.

Five participants dropped out of the study in problem set 2. They did so simply by doing some of their work outside Emacs or off the Athena computer system. We eliminated such participants — even those in the control group — because we were unable to measure their time worked and similar quantities, and could not account for effects that the other development environments (and switching between environments) may have had on their success.

We identified several problems with the tools (Section 5.5.1). Many of these have since been corrected, which would likely improve the results for the participants who were provided the tool. Furthermore, some students ignored the tools and thus gained no benefit from them. The results would probably be even better had those students used the tools.

We proposed continuous testing as an aid to a developer performing maintenance tasks such as adding functionality or refactoring in the presence of a regression test suite (though it may also be useful in other circumstances, such as test-first development). Continuous testing can be thought of as a way of making a test suite more valuable by using it more effectively: failing tests are discovered more quickly than they otherwise would be. Continuous testing is most useful when it relieves developers of difficult, lengthy, or easily forgotten tasks, and when developers are performing maintenance or other tasks that are likely to introduce regression errors. Because these circumstances were not present in our experiment, the experiment provided a much less than ideal scenario for a continuous testing tool. Testing was easy (it required only a single keystroke) and fast (a few seconds, see Figure 4); students were repeatedly reminded by the course staff to test often; and initial development tends to introduce fewer regression errors than does maintenance. These factors suggest that use of continuous testing in software maintenance (and in real project settings rather than the classroom) may yield even better results than measured by this experiment. As noted above, future experiments should build upon this one to evaluate continuous testing in other environments.

In our experiment, the developers were given a test suite ahead of time. Thus, our experimental results yield insight into the use of continuous testing in the increasingly popular practice of test-first development (and in certain other circumstances, such as when striving for compatibility or interoperability with an existing system). The experimental results are somewhat less applicable to maintenance, as noted above. Finally, they are not applicable to initial development in the absence of a test suite. Naturally, no test execution methodology can help in the absence of a test suite, or with errors that are not exposed by the test suite.

It is possible that a test execution methodology such as continuous testing, by focusing the developer's attention on the existing test suite, could distract the developer from considering other measures of progress for their project, such as conformance to a written specification. Our experiment does not shed light on this question, since it used the same definition of success that the students were judged on (passing all tests in the provided suite). One could run a similar study in which participants were required to pass an additional test suite that was not provided to them: if the results of the student solutions against the public suite and private suite matched, it would provide evidence that the solutions were generally correct, rather than simply "debugged into existence" against the public test suite. To suggest what the results of such a study would be, we created a new PS2 test suite containing 22 classroom tests of polynomial algebra and calculus [35]. The new test suite revealed only one new bug in one of the 17 student solutions.

7. RELATED WORK

We previously introduced the notion of continuous testing during development to reduce wasted development time [28]. The previous work also presented a model of developer belief that, along with a detailed record of a prior development project, enabled estimation of what the effects would have been, had the developer used a different testing tool in the prior project. A case study with one developer indicated that savings of 10–15% of development time could be possible. This research extends the previous research by implementing the continuous testing tool and performing a controlled experiment in order to measure rather than estimate the effect of the tool, and in order to obtain qualitative feedback regarding developer perceptions of the tool.

Continuous testing can be viewed as a natural extension of continuous compilation. Modern IDE's (integrated development environments) with continuous compilation supply the developer rapid feedback by performing continuous parsing and compilation, indicating (some) syntactic and semantic errors immediately rather than delaying notification until the user explicitly compiles the code. The Magpie [31] and Montana [14] systems pioneered practical incremental compilation to enable continuous compilation of large programs by only recompiling the part of a program which has changed, and this capability is standard in IDE's such as Eclipse and IDEA. Our study appears to be the first to empirically evaluate the productivity improvement provided by continuous compilation, but our programs were small enough that incremental compilation was unnecessary.

Continuous testing can also be viewed as a natural extension of Extreme Programming [2], which emphasizes the importance of unit test suites that are run very frequently to ensure that code can be augmented or refactored rapidly without regression errors.

Continuous execution [11], Programming by Example [6, 15], and Editing by Example [20, 17] all provide continuous feedback to developers about the results of their program on one or more inputs as the program changes. Our work abstracts from the entire output to the boolean result of each individual test case.

Several other authors use terms similar to our uses of continuous compilation and continuous testing. Plezbert [24] uses the term "continuous compilation" to denote an unrelated concept in the context of just-in-time compilation. His continuous compilation occurs while the program is running to amortize or reduce compilation costs and speed execution, not while the program is being edited in order to assist development. Childers et al. [5] use "continuous compilation" in a similar context. Siegel advocates "continuous testing", by which he means frequent manual testing during the development process by pairs of developers [32]. Perpetual testing or residual testing [23] (also known as "continuous testing" [33]) monitors software forever in the field rather than being tested only by the developer; in the field, only aspects of the software that were never exercised by developer testing need be monitored. Software tomography [22] partitions a monitoring task (such as testing [21]) into many small subpieces that are distributed to multiple sites; for instance, testing might be performed at client sites. An enabling technology for software tomography is continuous evolution of software after deployment, which permits addition and removal of probes, instrumentation, or other code while software is running remotely.

8. FUTURE WORK

Like any other experiment, ours has certain limitations (see Section 6). Future research should evaluate continuous testing in new situations, where it is as yet unproven. For instance, industrial case studies would provide additional qualitative information regarding continuous testing. We plan to provide continuous testing tools to a company performing software development for real customers, then observe and interview the developers to learn how they use the tools, their impressions of it, and their suggestions regarding it.

In this experiment, the test suites ran very quickly, easily providing real-time notification. There are several ways to extend this to suites that take longer to run.

First, we intend to integrate our Eclipse plug-in with one provided by Andreas Zeller that performs Delta Debugging [37]. Continuous testing gives early indication that a program change has introduced a regression error. However, when test suites take a long time to run, there may have been multiple program changes between the last successful test run and the discovery of a regression error. Delta Debugging can reduce the program changes to a minimum set that causes the regression error. Both continuous testing and this application of Delta Debugging reduce the number of program changes that a user must examine in order to understand the cause of a regression error. By using continuous testing, then Delta Debugging, the entire process might be made faster and more beneficial to the developer.

Second, just as continuous compilation on large programs is infeasible without incremental compilation (see Section 7), continuous testing on large test suites will require some form of incremental testing. For test suites with many tests, test selection [16, 10, 25] runs only those tests that are possibly affected by the most recent change, and test prioritization [36, 26, 34] uses the limited time available to run the tests that are most likely to reveal a recently-introduced error. We have taken some first steps toward showing continuous testing can be combined with some kinds of test prioritization [28], and are continuing to investigate how more traditional standard prioritization algorithms perform under continuous testing.

However, for tests suites with long-running tests, prioritization is insufficient: it will be necessary to use data collected on previous test runs to run only those *parts* of tests that may reveal recently-introduced errors, a technique we call *test factoring*. For example, if an error is introduced in the file input component of a compiler, a full end-to-end test is not necessary to find it — it will suffice to run only the file-input part of the test, and test that the generated data structures match what was observed in the previous run of the full test. We are actively investigating [29] the implementation of test factoring and its integration with continuous testing.

9. CONCLUSION

Continuous testing continuously runs regression tests in the background as the developer edits code and notifies the developer quickly when errors are discovered. To test the intuition that timely feedback is valuable for software developers (especially when the feedback is surprising), we augmented a development environment with continuous testing, and conducted a controlled experiment on student developers.

Developers with continuous testing were significantly more likely to complete the programming task than those without, without working for a significantly longer or shorter time. This effect could not be explained by other incidental features of the experimental setup, such as continuous compilation, regular testing, or differences in experience or preference. All developers in the study could manually run tests quickly with a single keystroke.

A majority of users of continuous testing had positive impressions, saying that it pointed their attention to problems they would have overlooked and helped them produce correct answers faster and write better code. Staff said that students quickly built an intuitive approach to using the additional features. 94% of users said that they intended to use the tool on coursework after the study, and 90% would recommend the tool to others. Few users found the feedback distracting, and no negative effects on productivity were observed.

Students who used continuous compilation without continuous testing were statistically significantly more likely to complete the assignment than those without either tool, although the benefit was not as great as that of continuous testing. Continuous compilation has proved a very popular feature in modern IDE's, but ours is (to our knowledge) the first controlled experiment to assess its effects. The results lend weight to the claim that it improves productivity, at least for some developers.

These positive results came despite some problems with the tools and despite continuous testing being used in a situation in which it does not necessarily give the most benefit: for initial development in a situation in which tests are easy to run and complete quickly. We have good reason to be hopeful that continuous testing will prove useful for many kinds of software developers.

Acknowledgments

We thank the students of MIT's 6.170 course who participated in our study, and the course staff who helped us to run it. Sebastian Elbaum and the anonymous reviewers provided helpful comments on the presentation. This research was supported in part by NSF grants CCR-0133580 and CCR-0234651 and by a gift from IBM.

10. REFERENCES

[1] V. R. Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of the 18th International Conference on Software Engineering*, pages 442–449, Berlin, Germany, Mar. 25–29, 1996.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[3] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, Boston, 2002.

[4] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 93–103, Portland, Oregon, May 6–8, 2003.

[5] B. Childers, J. W. Davidson, and M. L. Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 205–214, Nice, France, Apr. 22–26, 2003.

[6] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.

[7] A. Dunsmore, M. Roper, and M. Wood. Further investigations into the development and evaluation of reading techniques for object-oriented code inspection. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 47–57, Orlando, Florida, May 22–24, 2002.

[8] Eclipse. `http://www.eclipse.org`.

[9] Emacs. `http://www.emacs.org`.

[10] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[11] P. Henderson and M. Weiser. Continuous execution: The VisiProg environment. In *Proceedings of the 8rd International Conference on Software Engineering*, pages 68–74, London, Aug. 28–30, 1985.

[12] JetBrains IntelliJ IDEA. `http://www.intellij.com/idea/`.

[13] JUnit. `http://www.junit.org`.

[14] M. Karasick. The architecture of Montana: an open and extensible programming environment with an incremental C++ compiler. In *FSE '98, Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 131–142, Lake Buena Vista, FL, USA, Nov. 3–5, 1998.

[15] T. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *International Conference on Machine Learning*, pages 527–534, Stanford, CA, June 2000.

[16] H. K. N. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, Miami, FL, Oct. 16–19, 1989.

[17] R. C. Miller. *Lightweight Structure in Text*. PhD thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 2002. Also available as CMU Computer Science technical report CMU-CS-02-134 and CMU Human-Computer Interaction Institute technical report CMU-HCII-02-103.

[18] D. L. Moody, G. Sindre, T. Brasethvik, and A. Sølvberg. Evaluating the quality of information models: empirical testing of a conceptual model quality framework. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 295–305, Portland, Oregon, May 6–8, 2003.

[19] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, Nov. 20–22, 2002.

[20] R. P. Nix. Editing by example. *ACM Trans. Prog. Lang. Syst.*, 7(4):600–621, Oct. 1985.

[21] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Helsinki, Finland, Sept. 3–5, 2003.

[22] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 65–69, Rome, Italy, July 22–24, 2002.

[23] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 277–284, Los Angeles, CA, USA, May 19–21, 1999.

[24] M. P. Plezbert and R. K. Cytron. Does "just in time" = "better late than never"? In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 120–131, Paris, France, Jan. 15–17, 1997.

[25] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug. 1996.

[26] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.

[27] D. Saff. Automated continuous testing to speed software development. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, Feb. 3, 2004.

[28] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, pages 281–292, Denver, CO, Nov. 17–20, 2003.

[29] D. Saff and M. D. Ernst. Automatic mock object creation for test factoring. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, Washington, DC, USA, June 7–8, 2004.

[30] D. Saff and M. D. Ernst. Continuous testing in Eclipse. In *2nd Eclipse Technology Exchange Workshop (eTX)*, Barcelona, Spain, Mar. 30, 2004.

[31] M. D. Schwartz, N. M. Delisle, and V. S. Begwani. Incremental compilation in Magpie. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 122–131, Montreal, Canada, June 17–22, 1984.

[32] S. Siegel. *Object-Oriented Software Testing: A Hierarchical Approach*. John Wiley & Sons, 1996.

[33] M. L. Soffa. Continuous testing. Personal communication, Feb. 2003.

[34] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 97–106, Rome, Italy, July 22–24, 2002.

[35] Tacoma community college elementary algebra syllabus. `http://www.tacoma.ctc.edu/home/jkellerm/MATH090/default.htm`.

[36] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Eighth International Symposium on Software Reliability Engineering*, pages 264–274, Albuquerque, NM, Nov. 2–5, 1997.

[37] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 253–267, Toulouse, France, Sept. 6–9, 1999.

[38] M. K. Zimmerman, K. Lundqvist, and N. Leveson. Investigating the readability of state-based formal requirements specification languages. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 33–43, Orlando, Florida, May 22–24, 2002.