Reducing wasted development time via continuous testing

David Saff   Michael D. Ernst

MIT Computer Science & Artificial Intelligence Lab
200 Technology Square
Cambridge, MA 02139 USA
{saff,mernst}@csail.mit.edu

Abstract

Testing is often performed frequently during development to ensure software reliability by catching regression errors quickly. However, stopping frequently to test also wastes time by holding up development progress. User studies on real development projects indicate that these two sources of wasted time account for 10–15% of development time. These measurements use a novel technique for computing the wasted extra development time incurred by a delay in discovering a regression error.

We present a model of developer behavior that infers developer beliefs from developer behavior, and that predicts developer behavior in new environments — in particular, when changing testing methodologies or tools to reduce wasted time. Changing test ordering or reporting reduces wasted time by 4–41% in our case study. Changing the frequency with which tests are run can reduce wasted time by 31–82% (but developers cannot know the ideal frequency except after the fact). We introduce and evaluate a new technique, continuous testing*, that uses spare CPU resources to continuously run tests in the background, providing rapid feedback about test failures as as source code is edited. Continuous testing reduced wasted time by 92–98%, a substantial improvement over the other approaches.*

We have integrated continuous testing into two development environments, and are beginning user studies to evaluate its efficacy. We believe it has the potential to reduce the cost and improve the efficacy of testing and, as a result, to improve the reliability of delivered systems.

1. Introduction

Wasted time during software development costs money and morale. One source of wasted development time is regression errors: parts of the software worked in the past, but are broken during maintenance or refactoring. Intuition suggests that a regression error that persists uncaught for a long time wastes more time to track down and fix than one that is caught quickly, for three reasons. First, more code changes must be considered to find the changes that directly pertain to the error. Second, the developer is more likely to have forgotten the context and reason for these changes, making the error harder to understand and correct. Third, the developer may have spent more time building new code on the faulty code, which must now also be changed. If the error is not caught until overnight, these problems are exacerbated.

To catch regression errors, a developer can run a test suite frequently during development. After making a sequence of code changes, the developer runs the test suite, and waits for it to complete successfully before continuing. This synchronous use of the test suite leads to a dual inefficiency. Either the developer is wasting potential development time waiting on the CPU to finish running tests, or the CPU is idle waiting for the developer to finish a sequence of code changes, while regression errors potentially continue to fester, leading to wasted time fixing them down the road.

How can these two sources of wasted time be reduced? The testing framework with which the test suite is built could perhaps be enhanced to produce useful results in less time, by changing the way that errors are reported or the order in which they are run. The developer could test more frequently (catching regression errors more quickly at the expense of more time waiting for tests), or less frequently (reversing the trade-off), in hopes of striking a better balance between the two sources of wasted time.

Or the developer might use the test suite asynchronously, using the processor cycles unused during development to run tests. Without tool support, asynchronous testing means that the developer starts the test suite, and continues to edit the code while the tests run on an old version in the background. This is unsafe, however. If the test suite exposes an error in the code, it may be an old error that no longer exists in the current version of the code, and recently introduced errors may not be caught until the suite is re-run.

1.1. Continuous testing

We introduce here the idea of *continuous testing*, which uses real-time integration with the development environment to asynchronously run tests that are always applied to the current version of the code, combining the efficiency of asynchronous testing with the safety of synchronous testing. The developer never needs to explicitly run the test suite. The process can be tuned, prioritizing tests and parts of tests, to come as close as possible to presenting to the developer the illusion that the entire test suite runs instantaneously after every small code change, immediately notifying the developer of regression errors.

We conducted a pilot experiment to experimentally verify our intuitions about continuous testing before proceeding with our planned controlled user study. Are regression errors caught earlier easier to fix? Does continuous testing really promise a sizable reduction in wasted time, compared to simpler strategies like test reordering or changing test frequency?

To answer these questions, we needed both real-world data on developer behavior, and a model of that behavior that allowed us to make predictions about the impact of changes to the development environment. The data came from monitoring two single-developer software projects using custom-built monitoring tools to capture a record of how the code was changed over time, when tests were run, and when regression errors were introduced and later revealed by tests.

Our model, which is central to our analysis, is a finite automaton where states represent both whether the code under development actually passes its tests, and whether the developer believes that it does. Transitions among states are triggered by events such as editing code (to introduce or fix errors), running tests, and being notified of test results.

Used observationally, following the transitions triggered by observed events in the recorded data, the model can be used to infer developer beliefs about whether tests succeed or fail, distinguishing true accidental regression errors from intentional, temporary, changes to program behavior. Our analysis indicates a correlation between ignorance time (time between the accidental introduction of an error and its eventual discovery) and fix time (time between the discovery and correction of the error). This confirms that regression errors caught earlier are easier to fix, and permits us to predict an average expected fix time, given an ignorance time. We were then able to calculate that wasted time from waiting for tests and fixing long-festering regression errors accounted for 10% and 15% of total development time for the two monitored projects.

Used predictively, the model, together with the correlation observed above, can be used to evaluate the effect a change to the testing strategy or development environment has on wasted time. In this case, the observed events are altered to reflect a hypothetical history in which the new technique is used, and the model is re-run on the new event stream.

We evaluated three techniques for reducing wasted time. The first technique is to (manually) run tests more or less frequently. We discovered that the optimal frequency, for the projects we studied, would have been two to five times higher than the developer's actual frequency. The second technique is test prioritization, which reduces test-wait time by more quickly notifying developers of errors. In our experiments, the impact of test prioritization was non-negligible, but less than that of test frequency. The third technique is continuous testing, introduced above, which dominated the other two techniques and eliminated almost all wasted time.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents our model of developer behavior and beliefs Section 4 presents the quantities measured, the methodology used to gather data, and the specific development projects to which these methods were applied. Section 5 gives experimental results. Finally, Section 6 discusses future work to improve and further validate continuous testing, and concludes.

2. Related work

Modern IDE's (integrated development environments) for Java such as Eclipse and IntelliJ IDEA, offer *continuous compilation*. The IDE maintains the Java project in a compiled state as it is edited, speeding software development in two ways. First, the developer receives rapid feedback about compilation errors on every save, allowing for quick correction while that code is fresh in the developer's mind. Secondly, the developer is freed from deciding when to compile, meaning that when it is time to run or test the code, no intervening compilation step is necessary.

Henderson and Weiser [8] propose *continuous execution*. By analogy with a spreadsheet such as VisiCalc, their proposed VisiProg system (which they hypothesized to require more computational power than was available to them) displays a program, a sample input, and the resulting output in three separate windows. VisiProg treats a program as a data-flow-like network and automatically updates the output whenever the program or the input changes. Rather than continuously maintaining a complete output, which would be likely to overwhelm a developer, the test suite abstracts the output to a simple indication of whether the output for each individual test case is correct.

Programming by Example [5, 10] and Editing by Example [14, 13] can be viewed as varieties of continuous execution: the user creates a program or macro (possibly by providing input–output pairs), immediately sees its results on additional inputs, and can undo or change the program or macro in response. Our work differs in both its domain and in abstracting the entire program output to the test output, lessening the user's checking burden. (The user is required to write tests, which behave as a partial specification.)

The Extreme Programming methodology [2] 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 testing can be seen as taking this approach to its logical conclusion.

While it is desirable to run complete test suites, that may be too expensive or time-consuming. Regression test selection [11, 7, 18] and prioritization [23, 19, 22] aim to reduce the cost or to produce useful answers more quickly. Test prioritization is a key enabling technology for realistic continuous testing, and Section 5.4 compares several simple strategies. Our domain allows using data from previous runs of the test suite to aid prioritization, which we focus on here rather than data collected from coverage analysis of the program and test suite.

Kim, Porter, and Rothermel [9] examine the impact of testing frequency on the costs and benefits of regression test selection techniques, by artificially creating development histories that add defects to a working code base. Our work differs by using real development histories, and focusing on the impact on development time of changing test frequency and other techniques.

Boehm [3] and Baziuk [1] have shown that in projects using a traditional waterfall methodology, the number of project phases between the introduction and discovery of a defect has a dramatic effect on the time required to fix it. We are investigating whether similar results hold on the order of seconds rather than days.

Several other authors use terms similar to our uses of continuous compilation, continuous execution, and continuous testing. Plezbert [17] 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. [4] use "continuous compilation" in a similar context. Siegel advocates "continuous testing", by which he means frequent synchronous testing during the development process by pairs of developers [20]. Perpetual testing or residual testing [16] (also known as "continuous testing" [21]) 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 [15] partitions a monitoring task (such as testing) 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.

3. Model of developer behavior

Regression testing notifies a developer that an error has been introduced. If the developer was not already aware of the error, then he or she has the opportunity to correct it immediately or to make a note of it, rather than being surprised to discover it at a later time when the code is no longer fresh in his or her mind. (Notification is useful not only in cases where the developer wrongly believes that there is no error, but in cases where the developer believes there may be an error, but doesn't know which test fails, or why it fails.) If the developer was already aware of the error, then the notification confirms the developer's beliefs but does not affect his or her behavior. The notification is more useful in the former case, where the error was introduced inadvertently or unknowingly, than in the latter case, where the error was introduced intentionally or knowingly.

Assessing the usefulness of continuous testing for real development projects requires distinguishing between the two situations. Querying for the developer's beliefs regarding how many and which tests may be failing is distracting and tedious, affecting developer behavior and degrading quality of the answers.

Our approach is to unobtrusively observe developer behavior, then to infer, from developer actions, whether the developer believed a regression error to be present. This approach does not affect developer behavior, nor does it suffer from developer mis-reporting. The inferred beliefs are not guaranteed to be an accurate characterization of the developer's mental state; however, they do match our intuition about, and experience with, software development.

This section describes two models of developer behavior. The *synchronous model* describes the behavior of a developer who only gets test feedback by running the suite and waiting for it to complete before continuing development. The *safe asynchronous model* extends the synchronous model to describe developer behavior when test feedback on the current version of the code may be provided without the developer's explicit invocation during development (such as when the developer is using continuous testing). (We do not here consider an unsafe asynchronous model, in which a developer continues developing while tests run in the background on an old version of the code.)

Each model is a nondeterministic finite state machine. States indicate developer goals and beliefs about the absence or presence of regression errors; events (edges between states) are actions by the developer or the development environment. The synchronous model is a special case of the asynchronous model in which no notifications of test failures are ever received without explicit test invocation.

Both models have resemblances to the TOTE (Test-Operate-Test-Exit) model of cognitive behavior [12]. In the TOTE model, plans consist of an Image of the desired result, a Test that indicates whether the result has been reached, and Operations intended to bring reality closer to the Image. Here, running the test suite is the Test, changing the code to make it work is the Operation, and a successful test run allows Exit. The test suite is an externally communicated Image of the developer's desired behavior for the program.

A model of developer behavior can be used in two different ways: observationally and predictively. When used observationally, events (developer behavior and facts about the state of the code) drive the model into states that indicate the developer's beliefs. When used predictively, both states and actions are replayed into a model in order to infer developer behavior under different conditions. This paper uses the synchronous model both observationally and predictively, and the asynchronous model predictively.

Section 3.1 informally describes the synchronous model and the environment that it assumes. Section 3.2 presents the synchronous model for a test suite containing a single test. Section 3.3 extends the model to a test suite containing more than one test. Section 3.4 describes the safe asynchronous model.

3.1. Assumptions and terminology

The synchronous model applies to developers using a "guess-and-check" testing strategy. The developer changes the source code until he or she believes that the software is passing (the *guess* stage), then runs the test suite and waits for completion (the *check* stage). If the test suite fails, the developer has inadvertently introduced a regression error. The developer iteratively tries to fix the error and runs the tests, until no errors remain. Then the developer resumes changing the source code. We believe that this methodology is followed by many developers in practice. More relevantly, it was followed in our case study. Our models also account for the possibility that developers are following a test-first methodology such as Extreme Programming [2], in which a developer knowingly augments a working test suite with a failing test case, and then fixes the code to make the test pass.

The model assumes that the developer maintains an automated test suite that tests (part of) the software system's behavior. The developer may modify the source code or the test suite at any point in time. It is possible that the tests may be incomplete or incorrect with respect to some external specification of the program behavior, but these kinds of errors are not modeled. Our model ignores all points at which a test is unrunnable — for instance, due to a compilation error. A runnable test is either *failing* or *passing*, depending on what would happen if the test were run at that instant using the developer's current view of the code, including modifications made in unsaved editor buffers. A test that throws an unexpected runtime error is counted as failing.

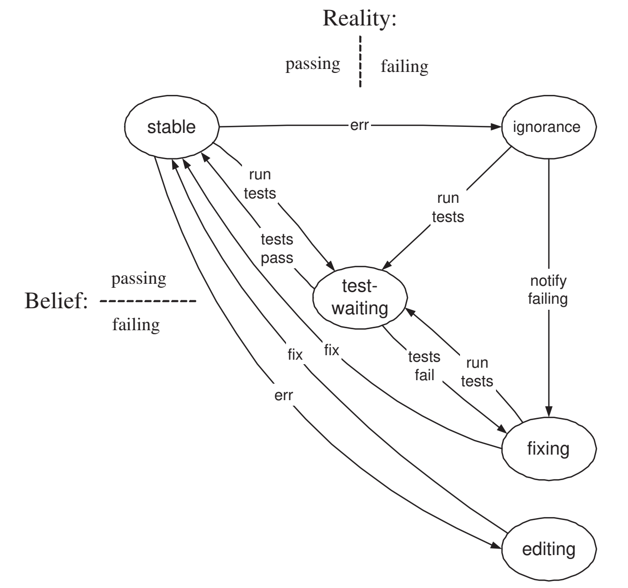

Figure 1. Nondeterministic finite-state automaton (NFA) for the synchronous model of developer behavior with a test suite consisting of a single test (Section 3.2).

3.2. Synchronous model for a single test

The nondeterministic finite-state automaton (NFA) shown in Figure 1 models the synchronous development process with respect to a single test case. The NFA has five states:

stable The test is passing, and the developer knows it. The developer is refactoring, adding new functionality, working on an unrelated part of the code, or not developing.

editing The developer has temporarily caused the test to be failing, and knows it. The developer is in the middle of an edit that is intended to make the test work again.

ignorance A regression error has been unknowingly introduced. The test is failing, but the developer does not know it. The developer continues to act as if the test is passing.

test-waiting The developer has started the test suite, and is waiting for it to finish.

fixing The developer knows (from a test failure) that a regression error has been introduced, and is working to fix it.

In the absence of continuous testing, six observable events map to transitions between the states in the model:

add test The test is first added to the test suite. This puts the model in the *stable* or *fixing* state, depending on whether the test is passing or failing. (For simplicity, Figures 1 and 2 omit these events.)

run tests The developer starts running the test suite.

test fail The test suite reports that the test has failed, together with details of the failure.

test pass The test suite reports that the test has passed.

err The developer (intentionally or inadvertently) introduces an error, making an edit that causes the previously passing test to fail.

fix The developer (intentionally) fixes an error, making an edit that causes the previously failing test to pass.

The synchronous model forbids some plausible behaviors. For example, it lacks a state corresponding to the situation in which the code is passing, but the developer believes it is failing. Although such a state would bring some symmetry to the model, we observed it to be very rare, and it does not capture any behavior that we wanted to consider in this research. As another example, the *fix* event always occurs with the developer's full knowledge. In reality, a regression error might be accidentally fixed, but we believe that such an event is uncommon and any such error is unlikely to be a serious one. As a third and final example, a developer only bothers to run the test suite when unsure about whether (some) tests fail, or when the developer believes that the tests pass and wishes to double-check that belief. The developer does not intentionally make a code change that introduces a specific regression error and then run the test suite, but without even trying to correct the error. (Note that this situation of causing an existing test to fail is quite different than augmenting the test suite with a new test that initially fails: we specially handle adding new failing tests, which is a common practice in test-first methodologies such as Extreme Programming.)

The reason for these restrictions to the model is two-fold. First, they make the model more closely reflect actual practice. Second, they enable resolution of nondeterminism. In Figure 1, the *err* event may happen with or without the developer's knowledge, transitioning from the *stable* state into either *editing* or *ignorance*. (This also explain the difference between the *editing* and *fixing* states.) The nondeterminism is resolved by whether the developer fixes the error before the next time the tests are run: a *fix* event is interpreted as meaning that the developer knew that an error had been introduced. A *run tests* event is interpreted as meaning that the developer thought there was no error the whole time.

3.3. Model for multiple tests

The model for a test suite containing multiple tests is built by combining the model for each individual test in the suite. Its state is determined as follows:

- If any test is in *ignorance*, so is the combined model.
- Otherwise, if any test is in *fixing*, so is the combined model.
- Otherwise, if any test is in *test-waiting*, so is the combined model.
- Otherwise, the combined model is in *stable*.

The multiple-test model has no *editing* state: nondeterminism is introduced and resolved at the level of individual tests.

The exact point when the model leaves the *test-waiting* state depends on the operation of the test harness that runs the test suite. If the harness waits until the end of all testing to give details about the success or failure of individual tests, then the model stays in the *test-waiting* state until the end of the entire suite run. If, however, the harness reports information on failures as soon as they occur, the model could immediately transition from *test-waiting* to *fixing*, on the assumption that the developer could immediately begin working on the new error before waiting for the tests to complete. For a passing run of the suite, we assume that the developer waits to see whether all tests have passed before continuing work. This indicates that even without introducing continuous testing, changes to a test suite or test harness may impact the speed of development. This notion is familiar from test selection and prioritization research; we explore this idea further in Section 5.3.

3.4. Safe asynchronous model

This section presents a model of developer behavior in an environment including continuous testing, which immediately displays feedback to the developer when a test fails. We use this model to predict the effectiveness of continuous testing.

The model of Figure 2 differs from that of Figure 1 only in adding a *notify failing* transition. This transition happens when the development environment notifies the developer that a particular test is failing, thereby ending the developer's ignorance about that error. If the notification happens in the *editing* state, it is confirmatory rather than providing new information — like to a word processor's grammar

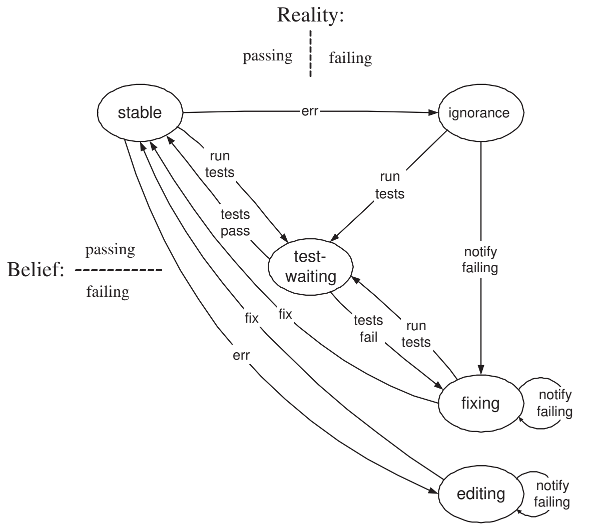

Figure 2. Nondeterministic finite-state automaton (NFA) for the continuous testing model of developer behavior with a test suite consisting of a single test (Section 3.4).

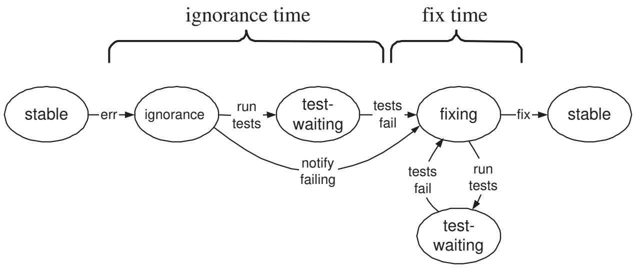

Figure 3. Definitions of ignorance time and of fix time. The diagram shows the transitions that occur when an error is inadvertently introduced, discovered, and then fixed. This diagram is based on the multiple-tests model of Section 3.3, which has no *editing* state.

checker flagging a subject-verb agreement error after the user changes the subject, when the user intended to change the verb next as part of a single logical edit.

Just as different test harnesses can cause different delays before the *tests fail* event, different test harnesses can cause different delays before the *notify failing* event.

The multi-test model is built from the single-test model as described in Section 3.3.

4. Experimental methodology

Using the model of developer behavior introduced in Section 3, we were able to define precisely the kinds of wasted time we wished to reduce (Section 4.1), develop an infrastructure for measuring them (Section 4.2), and apply this infrastructure to two development projects (Section 4.3).

4.1. Measured quantities

The key quantity that we measure is wasted time. We posit two varieties of wasted time: test-wait time and regret time. *Test-wait time* is the entire time spent in the test-wait state — reducing this time, all other things being equal, should lead to faster software development. *Regret time* is extra time that is wasted tracking down and fixing errors that could have been prevented on the spot with instant feedback, and fixing code based on a faulty assumption. Therefore, regret time manifests itself as increased *fix time*, the amount of working time between learning of an error and correcting it (see Figure 3). Some fix time is unavoidable: regret time must be inferred from *ignorance time*, the amount of working time spent between introducing an error and becoming aware of it.[1] The relationship between ignorance time and regret time may be computed for a particular development project (as we do in Section 5.1) by averaging over many observed instances to assign constants in a sublinear polynomial function. We calculate wasted time using a baseline of the predicted fix time for the minimum ignorance time observed, which is the lowest ignorance time for which it seems valid to make a prediction. This is a conservative estimate, and likely underestimates the possible benefits of continuous testing. To review, for a single error:

1. ignorance time = observed time in the ignorance state
2. predicted fix time = $k_1 *$ ignorance time ^{k_2}
3. regret time = predicted fix time[ignorance time] $-$ predicted fix time[minimum observed ignorance time]
4. wasted time = regret time $+$ test-wait time

To normalize test-wait time and regret time across projects, we express them as a percentage of the total time worked on the project. We estimated the total time worked by assuming that every five-minute interval in which the source code was changed or tests run was five minutes of work, and no work was done during any other times. This also applies to measuring ignorance time and fix times.

4.2. Implementation

We collected and analyzed data using two custom software packages that we implemented: `delta-capture` and `delta-analyze`.

1 As a simplification, we assume that the contribution of ignorance time to wasted time applies only to the immediately following fix time interval.

`delta-capture` monitors the source code and test suite, recording changes and actions to a log. `delta-capture` includes editor plug-ins that capture the state of in-memory buffers (in order to capture the developer's view of the code), and a background daemon that notes changes that occur outside the editor. The developer's test harness is modified to notify `delta-capture` each time the tests are run.

A *capture point* is a point in time at which a change has been captured. The interval between two consecutive capture points is a *delta*. The granularity of deltas is configurable by duration of time or number of keystrokes. A capture point also occurs immediately after each save and before any run of the test suite.

`delta-analyze` processes data collected by `delta-capture` in order to compute wasted time and determine how the programmer would have fared with a different testing strategy or development environment. First, `delta-analyze` performs *replay*: it recreates the developer's view of the source code and tests at each capture point, and runs each test on the recreated state. Second, `delta-analyze` uses the replay output, along with model of Section 3, to determine the state of the suite and to predict developer behavior, such as fix times. `delta-analyze`'s parameters include:

- A test-frequency strategy indicating how often the developer runs the test suite (see Section 5.2).
- A test-prioritization strategy that determines the order of synchronous tests run in the test harness and how the results are communicated to the developer (see Section 5.3).
- A continuous testing strategy (possibly none) that determines the order of asynchronous tests run during development (see Section 5.4).

4.3. Target programs and environments

We used an early version of `delta-capture` to monitor the remainder of the development of the tools of Section 4.2. The monitored development included a command-line installation interface, robustness enhancements, and code restructuring for `delta-capture`, and the creation of `delta-analyze` from scratch. Some of the code and test suite changes were in response to user comments from the 6 developers being monitored; this paper reports only data from the one developer who was working on the delta tools themselves, however. (Less data or lesser-quality data is available from the other half-dozen developers.) The developer used a test-first methodology. The test cases were not intentionally ordered in an attempt to catch errors early.

Because of different development environments, we separate the data into two groups and analyze them separately. The *Perl dataset* consisted of the development in Perl 5.8 of `delta-capture` and an early version of `delta-analyze`. The Perl tools were based on a shared library, and shared a single test suite. This project used the `Test::Unit` test harness and the Emacs development environment. The *Java dataset* consisted of a rewrite of most of `delta-analyze` in Java 1.4, using the JUnit test harness and the Eclipse development environment.

Attribute	Perl	Java
lines of code	5714	9114
total time worked (hours)	22	22
total calendar time (weeks)	9	3
total test runs	266	116
total capture points	6101	1634
total number of errors	33	12
average time between tests (minutes)	5	11
average test run time (secs)	16	3
mean ignorance time (secs)	218	1014
min ignorance time (secs)	16	20
median ignorance time (secs)	49	157
max ignorance time (secs)	1941	5922
mean fix time (secs)	549	1552
min fix time (secs)	12	2
median fix time (secs)	198	267
max fix time (secs)	3986	7086
max delta (secs)	15	60

Figure 4. Statistics about the Perl and Java datasets.

Table 4 shows statistics for the two monitored projects. Recall that time worked is based on five-minute intervals in which code changes were made. On average, the developer ran the Perl test suite every 5 minutes and the (faster) Java test suite every 11 minutes. The more frequent Perl test suite runs result from the developer's use of the Perl test suite to check the code's syntax as well as its functionality. The Eclipse environment provides real-time feedback about compilation errors, so the developer did not need to run the test suite to learn of syntax errors, type and interface mismatches, and similar problems.

5. Experiments

The data collected in the case studies of Section 4 allowed us to determine that a correlation exists between ignorance time and fix time (Section 5.1). This allowed us to compute the total wasted time for each project, which was 10% of total development time for the Perl project, and 15% for the Java project. We then used our developer behavior model predictively to evaluate three techniques for reducing this wasted time: more or less frequent testing by the developer (Section 5.2), test suite ordering and other test harness changes (Section 5.3), and continuous testing (Section 5.4).

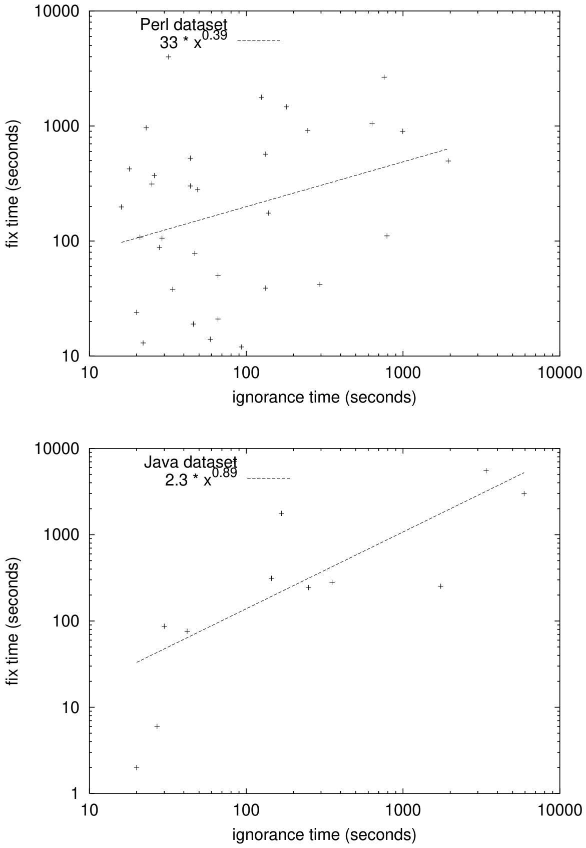

Figure 5. Scatter plot and best-fit line for fix time vs. ignorance time. Axes are log-scale, and the best-fit line is plotted.

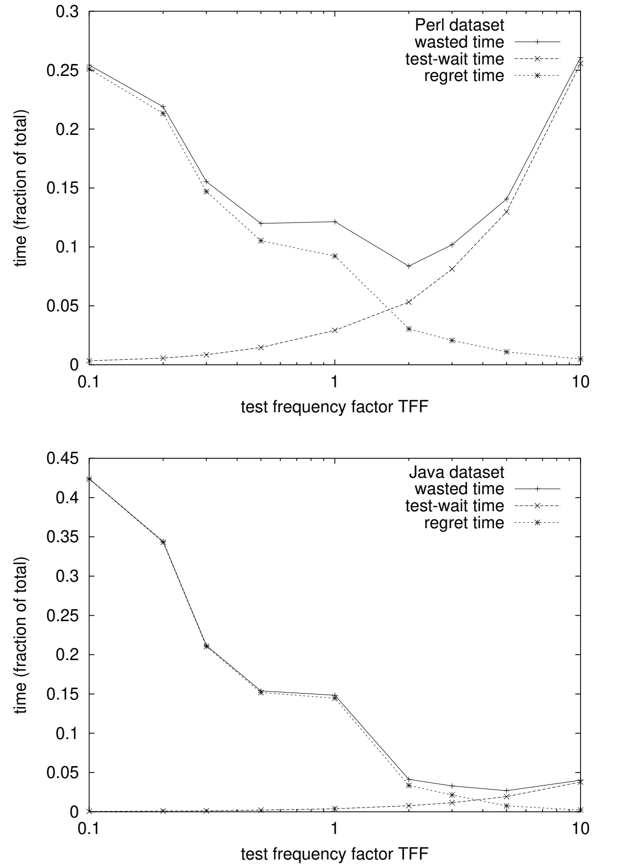

Figure 6. Wasted time as a function of testing frequency. Wasted time is the sum of test-wait time and regret time. The graphs show how increasing or decreasing the testing frequency by a multiplicative test frequency factor TFF affects wasted time.

5.1. Effect of ignorance time on fix time

Figure 5 plots the relationship between ignorance time and subsequent fix time. The plot uses the the multiple-test model of Section 3.3: if a change breaks multiple tests, the error introduced by that change is represented only once.

The figures also show the best-fit line relating ignorance time and fix time (the positive correlation has over 95% confidence in each case, supporting our hypothesis that reducing the ignorance time will speed development). A line on a log-log plot represents a polynomial relationship; the resulting polynomials are sub-linear (degree < 1; concave down). This relationship fits both the data and our intuition: the difference between 5 and 10 minutes of ignorance is more significant than the difference between 55 and 60 minutes of ignorance.

The relationship between ignorance time and fix time permits prediction of the impact on development time of changes to the development process. In particular, a tool that reduces ignorance time is expected to save the difference between the average fix time for the original ignorance time and the average fix time for the new ignorance time. Thus, we can fill in the constants in the formula for predicted fix time given in Section 4.1, as shown in Figure 5.

We treat the Perl dataset and the Java dataset separately because the different development environments, programming languages, and problem domains make errors easier or harder to introduce and to fix, yielding different relationships between ignorance time and fix time. Our subsequent experiments use the appropriate relationship to predict the effect of changes to the development process. As an example, for the Perl dataset, the relationship predicts that errors with an ignorance time of one minute take 3 minutes to correct, whereas errors with an ignorance time of one hour take 13 minutes to correct. For the Java dataset, the corresponding fix times are 2 minutes and 49 minutes.

5.2. Frequency of testing

Section 5.1 showed that ignorance time is correlated with fix time: reducing ignorance time should reduce fix time.

Developers do not need a tool to obtain this benefit: they can simply run their tests more often to catch more errors more quickly, at the expense of more test-wait time. (Continuous testing is like very frequent test execution, but without the test-wait penalty.) Running tests less often would involve the opposite trade-off. We investigated whether developers would be better off overall (and how much better off they would be) by increasing or decreasing the frequency at which they run tests. We simulated these behavior changes by adding and removing *test points*, which are capture points at which a run of the test suite was recorded.

Figure 6 shows the effect on wasted time as a developer tests more or less frequently. The test frequency factor TFF is the ratio of the number of simulated test suite runs to the number of actual test case runs. $TFF = 1$ corresponds to the observed developer behavior. For $TFF < 1$, we removed random test points; according to our model, test cases in the ignorance state at the time of a removed test point remain in that state until the next intact test point. For $TFF > 1$, we added random capture points to the actual test points; according to our model, when one of these new test points is encountered, any errors then in ignorance are transitioned into the fixing state. As always, regret time is computed from ignorance time using the formula given in Section 4.1, with constants derived from the data in Section 5.1.

As expected, increasing test frequency increases test-wait time, and decreases regret time. In the Perl dataset, the developer chose a fairly good frequency for testing, but could have reduced wasted time by 31% by testing twice as often. In the Java dataset, the developer could have enjoyed 82% less wasted time by running the tests 5 times as frequently— every 2 minutes instead of every 11 minutes. Here, a faster test suite and different development style mean that the increased test-wait time that results from more frequent testing is insignificant in comparison to the regret time, at least over the frequencies plotted.

5.3. Test prioritization and reporting

Changes to the test harness may save overall development time by reducing test-wait time. There are two related factors to consider. First is how long the test harness takes to discover an error, and second is how long the harness takes to report the error to the developer. Time for the harness to discover the error depends on the order in which tests are run. Figure 7 lists the test prioritization strategies that we experimentally evaluated. Time for the harness to report the error depends on its reporting mechanism. We considered three scenarios.

Full suite: The text harness runs to completion before reporting any results. This is the default behavior for the Perl `Test::Unit` harness. With this reporting mechanism, test prioritization is irrelevant.

Real-time: The test harness reports failures as they occur. This is the default behavior for the Java JUnit harness as integrated into Eclipse. Whenever a prioritization strategy listed in Figure 7 is mentioned, it is implicit that the harness reports failures in real time.

Instantaneous: All tests are run, and all failures are reported, instantaneously. With this reporting mechanism, test prioritization is irrelevant. This scenario represents the unachievable ideal.

Suite order: Tests are run in the order they appear in the test suite, which is typically ordered for human comprehensibility, such as collecting related tests together, or is ordered arbitrarily.

Round-robin: Like the suite order, but after every detected change, restart testing at the test after the most recently completed test. This is relevant only to continuous testing, not synchronous testing.

Random: Tests are run in random order, but without repetition.

Recent errors: Tests that have failed most recently are ordered first.

Frequent errors: Tests that have failed most often (have failed during the greatest number of previous runs) are ordered first.

Quickest test: Tests are ordered in increasing runtime; tests that complete the fastest are ordered first.

Failing test: The quickest test that fails, if any, is ordered first. This (unrealistic, omniscient) ordering represents the best that a perfect test prioritization strategy could possibly achieve. (Even better results might be achieved by introducing new, faster tests and prioritizing them well; see Section 6.1.)

Figure 7. Test case prioritization strategies used in the experiments of Sections 5.3 and 5.4.

Figure 8 shows, for each test harness and prioritization listed above, the effect of using it in terms of test-wait time, ignorance time, regret time, total wasted time (the sum of test-wait time and regret time), and improvement. Improvement is reported as reduction of wasted time and as a percentage reduction from the baseline measurement of wasted time.

For the Perl dataset, the recent errors test prioritization strategy dominated other achievable test prioritizations. In other words, the developer tended to break the same tests over and over. To our surprise, the frequent errors test prioritization strategy performed poorly. We speculate that this is because often a particularly difficult error persists for many test executions, thus dominating this metric due to test runs

Perl dataset:

Test Order	Test-wait	Regret	Wasted	Improvement	
Full Suite	0.055	0.044	0.100	0.000	0.0%
Random	0.038	0.042	0.080	0.020	19.6%
Frequent Errors	0.037	0.042	0.078	0.021	21.2%
Round-robin	0.029	0.041	0.071	0.029	28.9%
Suite Order	0.029	0.041	0.071	0.029	28.9%
Quickest Test	0.028	0.040	0.068	0.032	32.1%
Recent Errors	0.018	0.041	0.059	0.041	41.0%
Failing Test	0.018	0.041	0.059	0.041	41.1%
Instantaneous	0.000	0.039	0.039	0.060	60.5%

Java dataset:

Test Order	Test-wait	Regret	Wasted	Improvement	
Full Suite	0.005	0.145	0.150	0.000	0.0%
Frequent Errors	0.004	0.145	0.149	0.002	1.0%
Round-robin	0.004	0.145	0.149	0.001	1.1%
Suite Order	0.004	0.145	0.149	0.001	1.1%
Recent Errors	0.004	0.145	0.148	0.002	1.4%
Quickest Test	0.003	0.145	0.145	0.003	1.9%
Random	0.004	0.142	0.147	0.004	2.5%
Failing Test	0.003	0.142	0.145	0.005	3.6%
Instantaneous	0.000	0.142	0.142	0.009	5.7%

Figure 8. Effect of test harness and prioritization on the synchronous testing methodology, for the Perl and Java datasets. Each column is a fraction of total time (except percent improvement, which is based on the wasted time). Wasted time is the sum of test-wait time and regret time.

Perl dataset:

Testing Strategy	Test-wait	Regret	Wasted	Improvement	
Full Suite	0.055	0.044	0.100	0.000	0.0%
Random	0.038	0.002	0.040	0.060	60.2%
Frequent Errors	0.037	0.003	0.039	0.060	60.5%
Suite Order	0.029	0.003	0.032	0.070	67.8%
Round-robin	0.029	0.000	0.030	0.070	70.0%
Quickest Test	0.028	0.001	0.030	0.070	70.3%
Recent Errors	0.018	0.000	0.018	0.082	82.0%
Failing Test	0.017	0.000	0.018	0.082	82.4%
Instantaneous	0.000	0.000	0.000	0.100	100.0%

Java dataset:

Testing Strategy	Test-wait	Regret	Wasted	Improvement	
Full Suite	0.005	0.145	0.150	0.000	0.0%
Random	0.004	0.000	0.004	0.146	97.2%
Round-robin	0.004	0.000	0.004	0.146	97.3%
Suite Order	0.004	0.000	0.004	0.146	97.3%
Frequent Errors	0.004	0.000	0.004	0.146	97.3%
Recent Errors	0.004	0.000	0.004	0.147	97.7%
Quickest Test	0.003	0.000	0.003	0.147	98.0%
Failing Test	0.003	0.000	0.003	0.148	98.1%
Instantaneous	0.000	0.000	0.000	0.150	100.0%

Figure 9. Effect of continuous testing on wasted time. Each column is a fraction of total time (except percent improvement, which is based on the wasted time). Synchronous tests are executed at the same frequency used by the developer, using the same test harness strategy as the continuous testing strategy.

while in the *fixing* state, but may not be representative of regression errors discovered in the *ignorance* state.

For the Java dataset, none of the test prioritization strategies helped very much — they improved overall time by less than 1% and reduced wasted time by less than 4%. The reason is that the tests already ran very fast; little benefit could be had by reordering them.

5.4. Continuous testing

Using the same methodology as in Section 5.3, we evaluated the effect of various continuous testing techniques. Whenever `delta-capture` detects a change to the program or its tests (in other words, at each capture point that is not a test point), we simulate running tests until the next capture point. The full test suite may not be able to complete in this period, and we ignore any partially completed tests. This simulates a continuous testing environment that, every time the source is changed, restarts the test suite on the new version of the system, if the source is compilable. The same prioritization strategies listed in Figure 7 are used to indicate results in Figure 9. “None” indicates that continuous testing is not used; no *notify-failing* events occur.

Continuous testing using the recent-errors test prioritization reduces wasted time in the Perl project by 80%, and overall development time by more than 8%, as shown in the last two columns of Figure 9. The recent-errors test prioritization is nearly as good as the omniscient “failing test” strategy that runs only the fastest test that fails. Even using the (arbitrary) suite order saves a respectable 6.75% of development time. The frequent-errors prioritization performs nearly as badly as random ordering.

The improvement for the Java dataset is even more dramatic: Continuous testing improves development time by 14–15%, reducing wasted time by over 97%, regardless of the prioritization strategy, taking us near our conservative predicted limit for development speed-up from improved testing techniques. The prioritization strategies are essentially indistinguishable, because the test suite is quite small. We speculate that the greater gain for the Java dataset is due to the fact that the developer ran tests relatively infrequently. Section 5.2 showed that the developer could have reduced wasted time by 81% simply by running the tests more frequently. Together, these results indicate that continuous testing may have a substantial positive effect on developers with inefficient testing habits, and a noticeable effect even on developers with efficient testing habits.

6. Conclusion

6.1. Status and future work

We are currently setting up `delta-capture` monitoring for several additional software projects, which will help us to determine whether and how our results generalize to additional programmers, to larger projects and test suites, and to different development environments and styles.

Based on the encouraging results reported in this paper, we have integrated continuous testing plug-ins into both Eclipse and Emacs. Both environments unobtrusively indicate the existence of regression errors (via the Eclipse task list and the Emacs mode line) and permit the developer to click on the indicator to obtain more information about the errors. The plug-ins implement a variety of policies for recognizing when a change is complete, because testing an incomplete change could result in false positives. The Eclipse plug-in performs testing whenever the developer saves a file, and the Emacs plug-in performs testing whenever there is a sufficiently long pause; both plug-ins operate only if the code compiles correctly. We plan to use these plug-ins to perform both case studies, in which we observe how programmers interact with and benefit from environments including continuous testing, and also controlled experiments, in which separate developers (most likely college undergraduates) will perform a given task with and without the benefit of continuous testing.

We have also implemented an enhancement to the Perl `Test::Unit` test harness that has a real-time reporting mechanism and implements the Recent Errors test prioritization, based on the results in Section 5.3. This has been used in further development work on `delta-capture` and has proven very useful.

More feedback is not always better: an interface that provides too much information (particularly low-quality information) might interrupt, distract, and overload the developer, perhaps even to the point of retarding productivity. We believe that continuous testing can be implemented in a way that allows developers to take advantage of unobtrusive notification without being overwhelmed by it. In a survey of 29 experienced COBOL programmers, Hanson and Rosinski [6] found that they fell into two groups (of approximately equal size) regarding their tool preferences. One group preferred a larger, more integrated set of interacting tools; the other preferred to have fewer, more distinct, and less interrelated tools. This suggests that even if not all programmers embrace integration of continuous testing into their development environment, it will be valuable to at least some. Preliminary experience with our prototype suggests that many programmers appreciate continuous testing; we plan to perform additional studies to learn more.

Because our overall goal is to provide feedback, we intend to experiment with additional test prioritization strategies. Strategies based on code coverage seem particularly promising. After a developer edits a procedure, only tests that execute that procedure need to be re-run. This may greatly reduce the number of candidate tests, enabling getting to failing ones more quickly. Among other things, we will evaluate how much further prioritization can reduce ignorance and test-wait time, which are already fairly small even with simple prioritization techniques. We also must evaluate whether coverage information remains valid or useful as programs evolve.

Some test cases take a long time to run; this is particularly true of system or end-to-end tests. An environment that must wait for such tests to complete will not give the impression of instantaneous testing, even if it prioritizes test cases perfectly. Therefore, in addition to test prioritization, we are actively investigating *test factoring* to introduce new test cases that are smaller and faster. Test factoring determines how a system test uses a particular component, then creates unit tests for the component based on that usage. If (only) the component has recently changed, the unit test is just as effective as the system test, but more efficient. The unit tests can be made yet more efficient by eliminating redundancies.

6.2. Contributions

We have introduced the idea of continuous testing — using excess CPU cycles to test programs while they are being edited — as a feedback mechanism that allows developers to know more confidently the outcome of their changes to a program. Most significantly, it can inform them at an early stage of errors that might otherwise be detected later during development, when they would be more difficult and expensive to correct.

We have presented a conceptual model for investigating the usefulness of continuous testing and an experimental framework that implements the model. We ran experiments to evaluate the impact on development time of various continuous testing strategies. Together with additional results that verify that early detection of errors saves time overall, the results indicate that continuous testing has the potential to improve development, reducing overall development (programming and debugging) time by 8–15% in our case study. These are improvements to the critical resource in development: human time. Furthermore, the time saved is among the most annoying to developers: problems that could have been easily corrected earlier but grow in seriousness due to lack of attention. There is also promise that by increasing the usefulness of test suites, continuous testing will allow for a greater return on the investment of time put into producing tests.

In addition to evaluating continuous testing strategies, we have evaluated the development time reduction that could be achieved with a variety of test prioritization strategies. We have also shown how to assess whether a developer is running tests too frequently, too infrequently, or at just the right frequency. These techniques are in themselves effective: increasing test frequency for the Java dataset could reduce wasted development time by 81%, and a more effective test prioritization could produce a 41% reduction in wasted time for the Perl dataset. Our experiments also show which test prioritization techniques are most effective for developers in their daily practice; previous studies typically considered running regression tests just once, when a new version of a software system was complete. Continuous testing combines the best aspects of both test frequency and test prioritization. This is a promising field for further investigation, through additional monitoring, controlled experiments, and creation of effective new development tools.

Acknowledgments

We are grateful for the helpful comments of the referees, which improved the presentation of this paper. Derek Rayside provided valuable feedback on presentation. This research was supported in part by NSF grants CCR-0133580 and CCR-0234651 and by a gift from IBM.

References

[1] W. Baziuk. BNR/NORTEL: Path to improve product quality, reliability, and customer satisfaction. In *Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 24–27, 1995.

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

[3] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, 1976.

[4] 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.

[5] 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.

[6] S. J. Hanson and R. R. Rosinski. Programmer perceptions of productivity and programming tools. *Commun. ACM*, 28(2):180–189, Feb. 1985.

[7] 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.

[8] 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.

[9] J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 126–135. ACM Press, 2000.

[10] 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.

[11] 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.

[12] G. A. Miller, E. Galanter, and K. H. Pribram. *Plans and the Structure of Behavior*. Holt, Rinehart and Winston, Inc., 1960.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] 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.