From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing

Marat Boshernitsan
marat@agitar.com

Roongko Doong
roongko@agitar.com

Alberto Savoia
alberto@agitar.com

Agitar Software, Inc.
1350 Villa Street
Mountain View, CA 94041, USA

ABSTRACT

Developer testing is of one of the most effective strategies for improving the quality of software, reducing its cost, and accelerating its development. Despite its widely recognized benefits, developer testing is practiced by only a minority of developers. The slow adoption of developer testing is primarily due to the lack of tools that automate some of the more tedious and time-consuming aspects of this practice. Motivated by the need for a solution, and helped and inspired by the research in software test automation, we created a developer testing tool based on *software agitation*. Software agitation is a testing technique that combines the results of research in test-input generation and dynamic invariant detection. We implemented software agitation in a commercial testing tool called Agitator. This paper gives a high-level overview of software agitation and its implementation in Agitator, focusing on the lessons and challenges of leveraging and applying the results of research to the implementation of a commercial product.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging - testing tools.

General Terms

Algorithms, Performance, Reliability.

Keywords

Developer testing, unit testing, automated testing tools, software agitation, dynamic invariant detection, test-input generation, technology transfer.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

Copyright 2006 ACM 1-59593-263-1/06/0007 ...$5.00.

1. INTRODUCTION

Researchers and practitioners alike agree that developer testing is one of the most effective strategies for improving the quality of software and ensuring timely completion of software projects [19]. Developer testing requires developers to commit to testing their own code prior to integration with the rest of the system. Typically this type of testing is accomplished by creating and running small unit tests that focus on isolated behavior and do not require running the entire software system. Developer testing is not intended to replace system testing or integration testing that are typically practiced by the Quality Assurance (QA) organization. Rather, developer testing is designed to make it possible for the QA organization to focus on identifying system- and integration-level defects. This is critical for two reasons. The first reason is that the time and the cost of detecting and fixing a unit-level defect is orders of magnitude greater during system or integration testing than during development [32]. The second reason is that any time spent by the QA organization on unit-level defects either precludes or limits system- and integration-level testing.

The evidence suggests that the practice of developer testing finds limited and inconsistent adoption among developers. We believe that this stems both from a cultural problem—historically, developers have not been held responsible for testing their own code—and from a lack of adequate developer testing tools. Fortunately, there is reason for optimism. Agile software development practices, which have been growing in popularity in the last few years, consider developer testing an integral and important part of the development process [1]. A new breed of unit-testing frameworks (such as JUnit [13] and NUnit [15]) has helped to standardize the way unit tests are written and executed. But even with these frameworks, the cost of developing unit tests is high, both for individual developers and for development organizations. Tests have to be written by hand, and the combinatorial nature of software testing requires a significant amount of test code to achieve adequate application code coverage. It is difficult for developers to switch modes from development activities—mostly *constructive* and focused—to testing activities—mostly *destructive* and exploratory. Thinking about all the ways their freshly-written code could be wrong, broken, or incomplete is not natural for most developers. This is one of the most cited reasons for letting developers focus on development and QA engineers on testing.

A proper developer testing tool can address and minimize many obstacles to a more widespread adoption of developer testing. To achieve this, a developer testing tool must be effective in three areas. First, it must automate most of the tasks that don't require human intelligence, creativity, or insight. Second, it should facilitate and accelerate the developer's mental transition from development mode to testing mode. The tool has to present interesting input conditions that the developer may not have considered, and help the developer explore and discover the code's actual behavior under those conditions. Third, it should automatically generate a reusable set of tests to prevent regressions, once the developer is satisfied with the breadth and depth of exploration, has achieved a sufficient level of insight into the code, and has corrected any undesired behavior.

Various aspects of software test automation and the necessary static and dynamic code analyses have been popular research topics. The body of work in these areas already contains the core concepts and ideas necessary for building highly-effective developer testing tools. Yet, the benefits of much of this research have not reached the broader development community in a form that could be used in their day-to-day work. At Agitar Software, we leveraged several key test automation ideas from research, integrated them in a technique called *software agitation*, and built a novel developer testing product. This product, called Agitator, makes developer testing of Java programs practical and effective.

Software agitation brings two research ideas—dynamic invariant detection and test-input generation—to bear on the task of automating developer testing. Dynamic detection of likely program invariants was pioneered by Ernst in the Daikon invariant detector [5]. We adapted Ernst's work to the problem of developer testing. Automated generation, selection, and optimization of test-input values has been an active area in testing research. We combined some of the well-known ideas from that body of work with human-guided test-input creation to provide rich test inputs that enable automatic detection of invariants.

In moving these research ideas into practice, we faced many challenges. Dealing with these challenges involved understanding the trade-offs between different design issues such as completeness, correctness, usability, and performance. This paper presents some of these issues, summarizes our solutions, and discusses the lessons we learned while applying research ideas in the industrial setting.

The rest of this paper is organized as follows. In Section 2 we describe software agitation in more detail. We outline its research origins and summarize a number of contributing technologies that inspired the development of Agitator. Section 3 discusses some of the challenges that must be addressed in applying this research technology to "industrial-grade" software systems. Section 4 presents a high-level discussion of software agitation in Agitator. We describe the tool both at the architectural level and from the user's (developer's) perspective. We also present several of the lessons learned while building and deploying Agitator, including some of the surprising discoveries of how the use of Agitator affects software development habits. Section 5 summarizes our experience in using Agitator on itself. Section 6 presents some future directions and discusses a number of areas that warrant further research exploration.

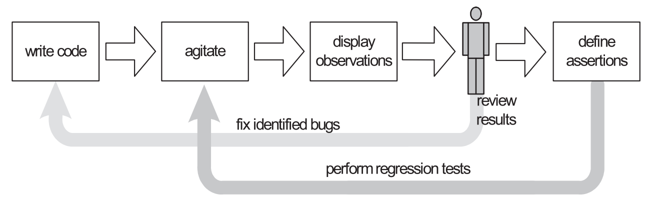


Figure 1: Overview of software agitation workflow.

2. SOFTWARE AGITATION

Software agitation is a unit-testing technique for automatically exercising individual units of source code in isolation from the rest of the system. Figure 1 presents a high-level view of the agitation workflow. Having finished a coding task, the developer invokes software agitation. The result of the agitation is a set of observations about the code's behavior. The developer checks observations to see if they reveal any bugs in the code and, if so, fixes those bugs and repeats the process. If an observation represents desired behavior, it can be "promoted" to an assertion. These assertions are checked during subsequent agitations to detect regressions from the desired behavior. Thus, software agitation does *not* entail test generation. Rather, it is a technique for exploring the implemented behavior of the code under test. When this behavior is desirable, it is the developer's responsibility to create an assertion (a test) that expresses that behavior. Software agitation reduces the grunt work, but it does not solve the test-oracle problem—this responsibility is delegated to the developer.

Broadly, software agitation comprises three phases: (1) creating instances of the classes being exercised, (2) calling all of the methods of those classes with a wide variety of input data, and (3) recording the results for subsequent analysis. The resulting *observations* represent the observed behavior of the unit under test. Observations take the form of relationships between various values in the source code that were determined to hold under a variety of different inputs. For example, for a method computing the maximum of two values a and b, agitation might determine that $max(a, b) \geq a \wedge max(a, b) \geq b$.

The development of software agitation was inspired by Ernst's work on dynamic invariant detection [5]. Dynamic invariant detection discovers program invariants from execution traces that are extracted by executing a system on a variety of valid test inputs. In their ICSE'99 paper [6], Ernst et al. demonstrate that Daikon dynamic invariant detection can infer many of the same invariants that a diligent programmer would write as part of a program's specification.

We extended the Daikon work to the domain of testing by noticing that dynamic invariant detection can be used for developer testing in two ways. First, the inferred invariants represent the observable behavior of the unit under test. Developers can check these invariants to determine whether the observed behavior coincides with the desired behavior. This facilitates early detection of defects. Second, developers can capture the invariants as part of the specification for the unit under test. We call this process "promotion of observations

to assertions." Subsequent agitations can check these assertions, creating the basis for automated regression testing. Developers can also add their own assertions to complement those discovered during agitation. These assertions become part of the testing suite.

One of Daikon's limitations is that it requires a broad range of input data values to generate a statistically significant sample of observed invariants. Automatic generation of test-input data has been a well-studied area of research. We combined several well-known techniques, such as symbolic execution, constraint solving, heuristic- and feedback-driven random input generation, and human input specification.

A similar application of dynamic invariant detection to testing was also independently proposed by Xie and Notkin in their ASE'03 paper [35]. (By that time Agitator was already in early beta testing, but not yet available to the public.) Xie and Notkin use an existing unit-test suite to exercise the program under test using Daikon, which was modified to generate design-by-contract (DbC) annotations for a commercial test generation tool. The DbC annotations represent pre- and post-conditions on individual methods in the program. The test generator attempts to generate additional unit tests that violate DbC invariants. Then, the developers examine the generated tests for potential inclusion in the unit-test suite.

Like software agitation, Xie and Notkin's tool is intended for interactive use. As in our approach, one of their major goals when generating test data is to challenge and violate assertions. Yet, unlike Xie and Notkin's approach, software agitation does not require any pre-existing tests. Software agitation also delegates more control to developers: it is up to the developer to decide which of the inferred invariants contribute to the specification.

Software agitation is a testing technique with a rich research heritage. It combines a number of well-researched ideas and applies them to the developer testing problem. In bringing software agitation to market we are indebted to the many researchers who have influenced and inspired our work.

3. CHALLENGES AND REQUIREMENTS

This section presents several design and implementation challenges that we faced while developing a commercial testing tool based on software agitation.

3.1 Usability

Software agitation involves several new concepts. Because the developer plays an essential role in the software agitation workflow, a software agitation tool must provide an interface that makes it easy to understand these new concepts and that fosters communication between a developer and the tool. This realization leads to several important conclusions.

Speak the developer's language. While it may be tempting to present discovered invariants to the developers in the form of axioms in first-order logic, it is likely that developers will not be able to connect the invariants to the behavior of their code. Likewise, we cannot expect developers to learn a new specification language or expend any significant effort on figuring out the tool.

These conclusions are partly due to Doong and Savoia's earlier work on the Assertion Definition Language (ADL) [23]. Despite its many innovations, the final report on the first four years of ADL research concludes that "developer productivity in producing [a test suite] was lower than would be expected from manual development [...] due to a very steep learning curve for ADL." [3]

Make no assumptions about developer's expertise. Some developers may have experience programming with invariants and pre- and post-conditions; others—may not. A software agitation tool must provide scaffolding for those developers that need it, while not inhibiting the work of more experienced developers.

Integrate into the established workflow. Software agitation requires developers to adjust some of their routines and habits. For example, when using software agitation the developers switch from a traditional edit/compile/debug cycle to an edit/compile/agitate/review/debug cycle. We discovered that it is essential for developers to be able to perform the agitate phase entirely within their preferred development environment. This implies that, in order to maximize adoption, a software agitation tool must be a fully integrated extension of the development environment and it must use the idioms and conventions of that environment.

Support refactoring. Helped by the automated support in most modern IDEs, code refactoring has become a very common practice. A software agitation tool must be able to track and accommodate refactoring operations to preserve developers' investment in assertions and test data.

Support mock objects. When the code under test relies on complex objects or layers of infrastructure, it is often impractical and time-consuming to set up an isolated environment for testing that code. Mock objects that conform to the interfaces of real objects but only implement partial functionality are an important component of a unit-testing strategy [30]. A software agitation tool must support and automatically generate mock objects when necessary.

3.2 Applicability

A software agitation tool must work with a variety of code bases. In contrast to the relatively small and self-contained examples described in published research, real-world applications are large collections of often nasty code. These application may have started with an elegant design but, over time, have organically evolved into systems that make testing a considerable challenge, even by a human. It is not unusual to find poorly-structured code where a single function or a method stretches for hundreds of lines. A software system may depend on a variety of support libraries for which the source code is not available, inhibiting source-level analysis. A Java application may include *native methods* implemented in a low-level language (typically, C or C++), inhibiting any analysis of those methods. Unfortunately, much of the existing research in automated testing makes simplifying assumptions about the amenability of a software system to analysis and about the size of these systems. A commercial software agitation tool must be ready to face these challenges.

Additional complications result from the continuous evolution of commercial software systems. From the day the first line of source code is written, the entire system un-

dergoes constant modification. Any tool that collects and stores information about a software system must be able to participate in this continuous change process. In particular, a developer testing tool must facilitate evolution of tests together with the system under test without requiring a developer to manually propagate source code changes to the tests.

3.3 Scalability and Performance

Software systems grow increasingly more complex and interdependent. It is not uncommon for modern software to consist of millions line of code written by a multitude of software developers over the course of many years. This complexity can affect the performance of any software development tool. A software agitation tool is no exception. While only a small part of source code is exercised during the agitation of a unit, many of the analyses that take place are global and non-linear with respect to the size of the source code base. Because software agitation exercises user code, bad performance of that code may reflect badly on the overall performance of the tool. At the same time, tight integration with the development workflow dictates that the response time of a software agitation tool must be sufficiently short to avoid distracting developers from their work.

These limitations present a serious dilemma. On the one hand, many of the recently developed program analysis techniques can be applied to software agitation to provide better test input data and to generate better invariants. On the other hand, less precise approaches can result in better performance on the real-life software systems. This choice, however, is not binary. Most of the analysis technologies can be outfitted with various shortcuts to tolerate some degree of imprecision and ambiguity. Finding a sweet spot in this continuum represents one of the most significant challenges in building a commercial tool for developer testing.

4. AGITATOR: THEORY INTO PRACTICE

This section briefly describes the implementation of software agitation in Agitar's Agitator. We make no attempt to provide a complete description of the product. Rather, we concentrate only on those aspects of the implementation that illustrate how we addressed the challenges in moving Agitator's research foundation into practice.

4.1 Agitator from a User's Perspective

In order to provide tight integration with the development workflow, we implemented Agitator's interface on top of Eclipse [4], a popular IDE for the Java programming language. While users can invoke agitation from the commandline, this mode of use is mostly reserved for build automation and is not meant for day-to-day development. Users invoke Agitator by selecting a unit to test (a class, a package, or even an entire project) and clicking the "Agitate" button. Starting an agitation is akin to invoking a compiler in a traditional development workflow. [1] Upon completion, Agitator presents several information panes (called "views" in Eclipse). Four of these views—the Outcomes view, the Observations view, the Snapshots view, and the Coverage view—are particularly illustrative of how Agitator meets the usability requirements put forth in Section 3. These views, shown in Figure 2, are discussed below.

The Outcomes View (Figure 2a). This view shows the *outcomes* that Agitator identified for a class or a method together with any additional *outcome partitions* that the user has defined. Outcomes represent possible ways in which a method's execution can terminate. Outcomes are determined statically—by considering all possible exit points from a method—and dynamically by observing the preceding agitation. A "normal" outcome occurs when no exceptions were thrown; other outcomes represent possible exceptions. The notion of method outcomes originated in ADL and represents a mechanism that enables users to reason about the flow of execution through a method. Users can further subdivide an outcome into *outcome partitions* that represent various post-conditions on that outcome.

Outcomes can fall into three categories: *expected*, *unexpected*, and *unspecified*. Expected outcomes include the normal outcome, any user-defined partitions of the normal outcome, and any explicitly thrown exception in the body of a method. Unspecified outcomes represent outcomes where the user does not care about tracking the results. Unexpected outcomes include Java runtime exceptions that are not explicitly declared in the method signature.

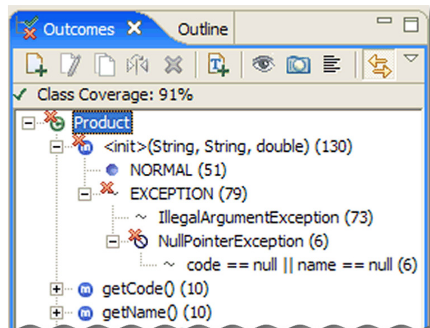
The Observations View (Figure 2b). This view displays *observations* and *assertions* for the selected class, method, or outcome. Observations represent the likely invariants inferred by Agitator during preceding agitation. Agitator displays separate sets of observations for each outcome (and for each outcome partition) of a method. When an observation represents expected behavior, users can click the check box next to the observation to promote it to an assertion. Assertions can also represent explicit conditions manually entered by the user. Each assertion gets a pass/fail score after each agitation run.

Agitator displays observations and assertions as boolean Java expressions. This choice of representation reflects our desire to present information to users in a language they can clearly understand. Using the expression notation from the underlying programming language raises the level of discourse between the user and the tool without overburdening the user. Similarly, the choice of terminology (e.g., "observation" *vs.* "likely invariant") is a conscious attempt to translate traditional notions into a more readily understandable form.

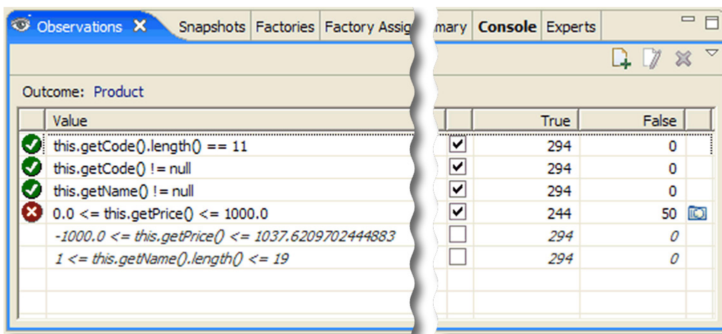
The Snapshots View (Figure 2c). This view shows sample input values used when exercising the method, for each outcome of a method. Each snapshot captures the state of the method's class and its parameters before and after a method call. The snapshots provide a mechanism for the user to examine the input data values used during agitation and to set debugger breakpoints based on these values.

The Coverage View (Figure 2d). Agitator presents coverage information on a side panel inside the Eclipse source code editor. Coverage reflects Agitator's success in finding data values that direct execution to every statement and to every possible exception in the code. The coverage metrics used by Agitator represent line coverage, augmented with

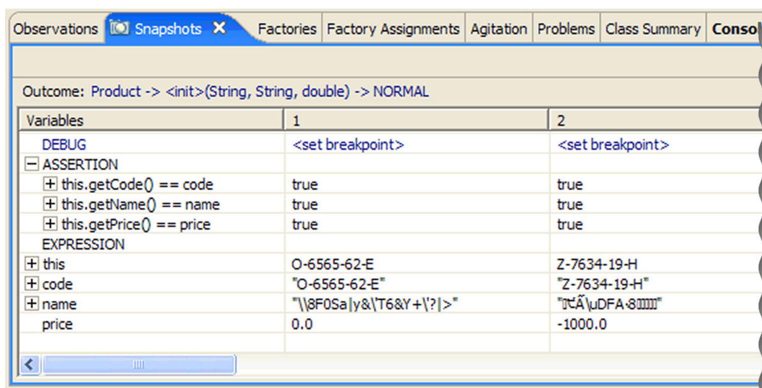
[1] The compiler metaphor is not entirely accurate in the case of Eclipse, where incremental compilation (when enabled) eliminates a separate compilation step.



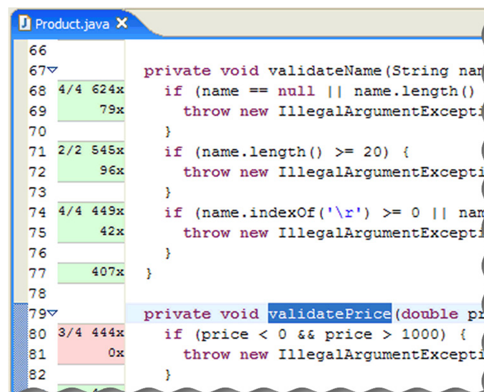
(a) Outcomes View



(b) Observations View



(c) Snapshots View



(d) Coverage View

Figure 2: Four information views available to Agitator users upon completion of agitation.

expression coverage for every conditional in a branch. Users use the coverage view to assess the quality of the agitation and to decide whether they need to assist Agitator in finding suitable input values to improve coverage.

The coverage view represents one of the mechanisms that Agitator uses to provide immediate feedback about agitation to users. By reviewing the sections of source code that were not covered during agitation, users can identify input values that Agitator was not able to construct, and detect possible unreachable paths in the source code.

4.1.1 *Helping Agitator Generate Input Data*

Using auto-generation of input data, Agitator can generate many relevant test-input values with no user intervention. For some methods, however, users need to refine the generated values by specifying a valid range of input data or by providing a set of test values that improve the quality of observations. In some cases, it may be necessary to supply more information so that Agitator can construct complex object states. (This situation is further considered in Section 4.3.2, where we discuss the analysis algorithms used to construct test-input data.) In all of these cases, Agitator allows users to define *test-input factories* to accomplish their goals.

A factory tells Agitator how to construct interesting and relevant test objects of a given type. Users can define factories by configuring one of the predefined factories, or or by using the Factory API to define their own. Agitator includes a rich set of pre-defined factories. These factories range from the very simple, such as those that return values in a given numerical range, to the fairly sophisticated, such as those that use reflection to create complex objects from constructors or methods. Factories are composable. For example, users can configure an integer factory to generate values within a specific range, and then use that factory in a collection factory to generate collections of integers in that range. The factories defined by one user become durable assets for the rest of the development team to use and share.

4.1.2 *Domain Knowledge for Testing*

Getting the most out of Agitator on industrial-scale source code bases requires some amount of "tweaking." This is especially true when an application utilizes a third-party framework (such as J2EE [26] or Struts [27]) that requires some configuration to enable independent agitation of units. In such situations, many reusable test assets such as factory configurations, outcome partitions, and assertions are common across all classes and methods that rely on these frameworks.

Configuring Agitator for every such application requires domain knowledge and understanding. To facilitate and accelerate this task, Agitator enables packaging of common domain knowledge in independent modules that can be applied in various application contexts. These modules are

called *domain experts*. Agitar provides experts that address some common patterns, such as those arising in J2EE [26], Struts [27], Hibernate [11], and Spring [25].

Agitator recognizes code patterns and uses experts to enable better agitation results by applying appropriate boilerplate configuration that provides factories, assertions, and outcome partitions. Some experts are applied automatically; others are configured by the user. Using experts eliminates many of the repetitive tasks that users would otherwise have to perform when agitating source code that uses a common framework.

New experts can be created using the Expert API. The Expert API is not intended for everyday use. Rather, this facility exists for the benefit of framework and component providers, who wish to supply an Agitator expert with their library. We envision the Expert API as evolving into a generic plug-in mechanism for Agitator, enabling much more extensive use of the experts within the internals of the agitation process. This would enable the use of Agitator as a platform for research into novel techniques and methods for automated testing. We present this vision in more detail in Section 6.

4.2 Agitator-Driven Refactoring

Developers often modify their work habits to incorporate a new tool into their workflow. For example, Saff and Ernst [22] report that the users of their continuous testing tool changed their working style to ensure that they "got a small part of [their] code working before moving on to the next section." Similarly, users of Agitator found that following certain coding patterns not only leads to better designed code, but also enables Agitator to provide better results because the code is more testable and the tests are more thorough and more readable. One example of this is what we call *Agitator-driven refactoring*.

The concept of *refactoring* was introduced by Opdyke [16] and involves modifying software source code to improve some aspect of its internal structure without affecting its behavior. Fowler's book [7] catalogs many refactoring transformations that have since become standard practice. Many of these transformations have the beneficial side effect of helping Agitator users to affect some aspect of agitation.

Introduce Type. This refactoring introduces a new class that wraps a primitive type (such as `int`) or a non-specific class (such as `java.lang.Integer`).

Agitator works best when it has the most information about the format of the data that it is trying to use as input to agitation. Unfortunately, it is a common practice to conflate various types of values by using primitive or insufficiently specialized types. For example, a salary value may be represented as an integer, carrying no indication that the salary may not be negative. Moreover, nothing in such a representation distinguishes a salary value from an employee number, permitting Agitator to reuse a value obtained by calling `getSalary()` as an input to `setEmployeeNumber()`. The *Introduce Type* refactoring alleviates this problem by encapsulating each value in the corresponding class, such as `Salary` or `EmployeeNumber`.

Replace Error Code With Exception. This refactoring replaces the return of an error code (such as `return -1`) with the throw of an exception (such as `throw new IllegalArgumentException`).

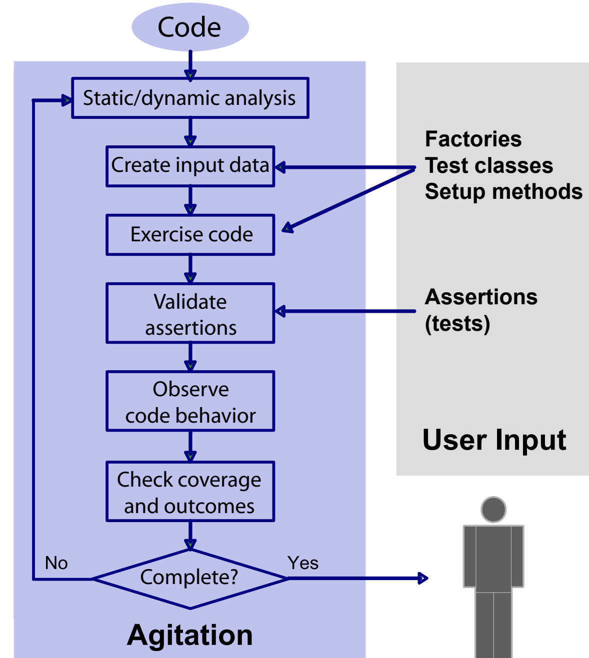


Figure 3: Inside agitation.

Agitator uses exceptions thrown from the source code to create distinct outcome partitions for each exception. This permits easier tracking and debugging of the exception outcomes. Similar results can be achieved by manually constructing an outcome partition that spells out the conditions under which the error code is returned. Having Agitator create an exception partition automatically, however, is much easier for the user.

Extract Method. This refactoring is used to extract a fragment of an existing method's body into a new method. Extracting a method permits giving a descriptive name to the method and its parameters.

Agitator has a better chance of achieving coverage and generating good input data values for methods with simpler control flow. While it works on long "spaghetti-code" methods, the list of observations generated from smaller methods is usually shorter and more relevant.

4.3 What Happens During Agitation

The agitation process includes both automated processing and human intervention. This section focuses on the internals of the agitation process and on the individual steps that constitute agitation, as presented in Figure 3.

4.3.1 Analysis and Input Data Generation

Software agitation is an iterative process, driven by the objective of reaching maximum code coverage while achieving every possible expected outcome. Software agitation begins with static and dynamic analysis of Java code (at the bytecode level), followed by dynamic generation of test data based on identified behavior of the code. The dynamic analysis utilizes data values and coverage information from

the preceding iteration to direct execution along a particular code path. The static analysis uses heuristic-driven path analysis to collect and solve input value constraints that can cause execution to take the selected path.

In order to satisfy the scalability and performance requirements described in Section 3, Agitator takes several shortcuts and makes a number of approximations in the analysis algorithms. For instance, rather than trying to solve constraints for the entire execution path, Agitator may consider only a part of that path. While such an approach may not always direct execution down the expected path, we found that it works well in practice and cuts down analysis time. Similarly, Agitator does not perform a receiver-class analysis when considering paths spanning multiple methods. Instead, it uses heuristics to "guess" the potential candidates among all possible implementations of a method. The heuristics used by Agitator were determined mostly via profiling and experimentation. For example, Agitator unrolls loops and recursive calls until the control-flow graph reaches a particular size. We determined (experimentally) that this size gives a good trade-off between the completeness of the results and Agitator's performance.

Solving of path constraints is implemented in Agitator using a wide array of generic and specialized constraint solvers. For instance, Agitator includes a string solver for `java.lang.String` objects, producing strings that satisfy constraints imposed by the String API (e.g., a string that `.matches(...)` a regular expression). Similarly, Agitator includes several solvers for the Java Collections classes.

In addition to using constraint solving and other sophisticated methods for generating test inputs, Agitator uses a few heuristics that often allow it to explore otherwise unreachable paths. Some of these include:

- For integers: use 0, 1, -1, any constants found in the source code, the negation of each such constant, and each constant incremented and decremented by 1.
- For strings: generate random text of varying lengths, varying combinations of alpha, numeric, and alphanumeric values, empty strings, null strings, any specific string constants occurring in the source code.
- For more complex objects: achieve interesting object states by calling mutator methods with various autogenerated arguments.
- For all objects: use various objects of the appropriate type created during execution. Agitator accumulates some of the objects that it creates at various points during agitation for possible reuse in place of constructing new ones.

In some cases, Agitator lacks the information to construct an object, or to put an object in the specific state required to test a class. In those cases, users provide help by defining one or more factories that tell Agitator how to construct an object (see Section 4.1.1).

The above shortcuts represent a point of departure from the established research in test-input generation. Most of these were necessary to achieve adequate memory and runtime performance. An important benefit afforded by the iterative nature of software agitation is that it is not necessary for Agitator to achieve the desired goal on the first attempt. Moreover, because Agitator must execute every path multiple times (see the next section), it is perfectly acceptable for the analysis algorithms to "guess" a good set of data values and to adjust that guess if the execution does not take the expected path. Guessing the right mix of data values entails approximating an equivalence partition for each input parameter and trying different combinations of values in that partition. In order to facilitate invariant detection, Agitator attempts to provide five distinct data values for every input parameter for an execution path. Thus, the goal of test-input generation is to create a large number of input values that maximize the likelihood, rather than guarantee that these values provide the desired coverage.

Further research is needed to adapt some of the recent work in test-input generation to the commercial-scale source code bases. We revisit this point in Section 6.

4.3.2 Execution and Invariant Detection

After test-input generation, Agitator exercises each method in each class under test. Agitator attempts to execute every path through a method multiple times, so that the invariant inferencing algorithm has sufficient distinct data points. Agitator uses heuristics to decide whether it executed a particular path enough times with sufficiently distinct data values. Infinite loops or long-running operations are terminated using user-settable timeouts.

Execution of potentially defective source code with random inputs may present a significant risk to system security and integrity. For example, a utility that deletes files from the filesystem must not be allowed to do so during its execution by an agitation tool. Agitator takes special care to isolate all application source code that it executes using the Java security manager.

During agitation of each testable method, Agitator applies its dynamic invariant detection algorithm to discover "interesting" method invariants that can be presented to the user as observations. The algorithm for invariant detection in Agitator is different from that used in Daikon. The Daikon approach is to record a detailed execution trace that captures every variable value at every appropriate program point. Daikon subsequently analyzes this execution trace and performs invariant inference over the captured data values. The inferred invariants are further filtered to remove those that present no interest to the user (for more information see Chapter 5 of the Daikon User Manual [28]).

Again, due to the scalability and performance considerations, we had to adopt a different strategy for invariant detection. Rather than inferring invariants from individual data values after agitation, Agitator starts out by positing a number of hypothetical relationships between values within a method. Because the number of possible combinations can be very large, Agitator filters the list of hypothetical relationships using simple heuristics. For example, we assume that two variables occurring in the same expression are related, whereas variables that never occur in the same context are independent. During agitation, Agitator discards relationships that do not always hold. When agitation is complete, the list of "surviving" relationships is presented to the user as observations.

Agitator finds data values to observe using the following heuristics:

- By examining the values of method parameters and instance variables.
- By checking properties of simple data types (array length, string length, etc.).
- By identifying "getter" methods for objects in scope.
- By considering fields related to the method under test.
- By including initial values (values prior to execution) for all observable expressions in a method and by considering the return value of that method.

When forming a hypothesis about data value relationships, Agitator considers a number of properties such as:

- **Expressions in source code.** Simple expressions found in the method under test are always observed and checked for invariability.
- **Relational: $x = y$, $x > y$, $x \geq y$, $x < y$, $x \leq y$.** These relationships are possible between numeric data types. Note that with our approach, hypothesis $x \neq y$ is useless (unless it occurs in the code), because the likelihood of disproving it is small.
- **Range: $A < x < B$, $A \leq x < B$, $A < x \leq B$, $A \leq x \leq B$.** Another set of relationships for numeric values. Agitator makes initial estimates for the values of A and B and then iteratively refines the boundaries.
- **Logical: $x \wedge y$.** Agitator checks this relationship for boolean values. Early versions of Agitator also checked $x \vee y$ and $x \Rightarrow y$, but we found that considering these relationships resulted in too much noise (useless observations).
- **Linear: $Ax + By = C$, $Ax + By + Cz = D$.** Agitator checks these relationships for integer values. The three-variable relationship is only considered when A, B, and C are $+1$ or -1; the two-variable relationship is checked for all constant values.
- **Object: $x = y$, $x.equals(y)$, $x = null$, $x \neq null$.** Agitator considers these relationships for all applicable object values.

Agitator considers relationships that remain after agitation invariant, sorts them by relevance, and presents them to the user. The relevance score reflects how closely components of the relationship are "related" in the source code. Expressions that appear in source code verbatim score the highest; expressions that are pure guesses score the lowest.

Our other point of departure from the Daikon approach is to preserve implied invariants, rather than attempt to eliminate them. (The latest releases of Daikon even include the interface to a theorem prover specifically for that purpose.) Our reasons for keeping redundant invariants are twofold. First, we found (via experimentation) that eliminating implied invariants proved too taxing on Agitator's performance. Because the tool is used interactively, reducing the response time is essential for usability. Second, we discovered that a fully reduced set of observations was not always appropriate for human inspection. For example, consider the following two sets of observations (full and implied):

$$\begin{array}{rcl} x & = & y \\ y & = & z \\ x & = & z \\ x & \neq & null \\ y & \neq & null \\ z & \neq & null \end{array} \quad \Rightarrow \quad \begin{array}{rcl} x & = & y \\ y & = & z \\ x & \neq & null \end{array}$$

Logically, these two sets are equivalent. The user, however, might want to see and test each assertion separately and not hide the observation $z \neq null$ inside the deductive chain ($z = y \wedge y = x \wedge x \neq null \Rightarrow z \neq null$). Consider another example:

$$\begin{array}{lrcl} (1) & x & > & y \\ (2) & x & = & y+1 \end{array}$$

Here, we do not know whether the right behavior is (1) or (2). Perhaps, the intended behavior is (1), but due to a bug in the user's program, the user discovers that the code always increments by 1. By presenting both options, Agitator shifts the responsibility for this decision to the user, where we believe it belongs.

5. AGITATING AGITATOR

After 18 months of development Agitator had enough functionality to be used on its own code. In this section we report some results of running Agitator on itself. Most of the metrics reported here are publicly available as part of Agitar's Open Quality initiative, an effort to demonstrate the quality of software via auto-generated publicly-viewable reports.[2] These reports include code coverage achieved both by manually-written unit tests and by automatic agitation, the number of assertions, and other project code metrics. The information we present corresponds to Agitator 3.0.

5.1 Project Statistics

Agitator is implemented in Java. It consists of 1,734 application classes and 1000 unit-test and test harness classes. Agitation, including execution of manually-written unit tests, achieves 80.2% code coverage. During agitation Agitator checks 30,998 assertions. This includes both the assertions in the manually-written tests and the observations promoted to assertions. There are 68,990 executable lines of application code and 29,404 executable lines of unit-test and test harness code. Agitation is supported by 26 custom test-input factories that help Agitator achieve adequate code coverage. The implementation of these factories takes another 766 executable lines of code. Taken together, these metrics indicate test code to application code ratio of 1:2.3 and assertion to code ratio of 1:2.2.

Comparing these statistics to other projects is challenging because very few publicly available software systems include unit tests that achieve comparable code coverage. Some systems, however, are better at this than others. For instance, consider the Commons-collections library [29], part of the Jakarta project. As of version 3.1, this library includes unit tests that result in 77.7% code coverage. The ratio of test code to application code, however, is significantly larger—1.62:1. This means that on average it takes 1.62 lines of test code to test a single line of application code, compared to 1

[2] http://www.agitar.com/openquality/

line of test code per 2.3 lines of application code in Agitator. The ratio of assertions to application code is comparable to that of Agitator—1 assertion per 2 lines of source code. This comparison demonstrates that the use of Agitator can reduce the effort by requiring less test code to achieve comparable application code coverage. Of course, some effort must be spent on checking Agitator's observations, possibly promoting them to assertions, and on manually adding new Agitator assertions in the observation view. We look at this process next.

5.2 From Observations to Assertions

To measure Agitator's efficacy in helping Agitator developers we estimated the number of "useful" observations that it produces from its own source code. We define "usefulness" very narrowly, counting only those observations that the developers promoted to assertions *without editing* to make them stronger or more meaningful. Furthermore, we looked only at a single point in time and only at the observations that existed at that point. Because we use agitation continuously during the development process, many observations that are examined by the developers represent unexpected behavior in the code. These observations are clearly "useful" because they help to fix bugs, but they are not included in our computation.

In the source code for Agitator 3.0, Agitator is able to exercise 34,004 method outcomes. In doing so, it creates 126,374 observations. Over the time that Agitator has been self-testing, 13,868 of these observations were promoted to assertions. This indicates that at least 11% of Agitator's observations represent useful invariants that have been promoted to assertions. The total number of assertions that Agitator checks is actually higher—18,623. Some of the extra assertions resulted from observations that Agitator developers modified to make them stronger or more meaningful. Some of these assertions developers entered manually. Because we have not preserved the distinction between these two kinds of assertions they are excluded from our estimate.

6. THE ROAD AHEAD

After two years of commercial deployment we are happy to state that Agitator has become an invaluable tool for many of our customers. Still, much work lies ahead. Some of the challenges are purely technical; others—require more research in various areas, ranging from better test-input generation algorithms to understanding developers' motivation for writing tests. In this section, we outline several directions for further exploration.

6.1 Agitator as a Research Platform

Creating a research infrastructure is the bane of many software engineering research projects. On the one hand, the infrastructure is necessary to provide the support services, such as some basic source code analyses. On the other hand, building such an infrastructure typically involves a significant engineering effort by several researchers, while providing little opportunity for novel research.

We would like to extend Agitator to provide such a platform for research in testing tools. Even now, the Expert API (Section 4.1.2) provides some rudimentary hooks to Agitator's internals. We are interested in extending this API and creating a true plug-in architecture that would enable outside researchers to modify and extend Agitator's behavior.

6.2 Improving Testing Technology

In the recent years, test-input generation has been the area of active research. Advances in techniques for symbolic execution (e.g., [31, 34, 24, 33, 20]), random testing (e.g., [8]), and test selection (e.g., [17, 36, 9]) suggest many plausible improvements to the algorithms currently used by Agitator. Yet, scalability of these approaches remains a major obstacle to adoption. Further research is needed in adapting these and similar techniques to large, commercial-scale code bases. One promising direction, suggested by Tillmann and Schulte [31], is to store the intermediate results of the analysis of common (library) code in a reusable manner. Sen [24] proposes a set of domain-specific optimizations that improve the performance of constraint solving. Robschink and Snelting [21] show how to solve path constraints efficiently in realistic-size programs by using interval analysis and binary decision diagrams.

Increasingly, we are seeing software systems that are implemented using several programming languages. For example, some of the Java libraries include native code components implemented in C and C++. Unfortunately, we are not aware of any research that attempts to address analysis and generation of test inputs across the boundary between two (or more) languages.

6.3 Helping the User

The human component plays a big role in software agitation. While we believe that we developed a successful user interface for software agitation, our work is far from complete. Below are just a few of the ideas that we feel merit further exploration.

Visual language for factory configuration. Input-value factories in Agitator are configured using interactive dialogs. Often, the factories need to nest (for example, an array factory that uses another factory for its components), sometimes more than a single level. Configuration of nested factories can be confusing to the user. The dependencies between different factories can be visualized using a dataflow network, similar to those used in visual programming languages, such as SCIRun [18] and Fabrik [12]. Designing a specialized visual programming language for factory composition and configuration would be a welcome improvement to Agitator.

Assertion generalization. Observations reported by Agitator are fairly low-level, code-based invariants. We are interested in exploring mechanisms that can generalize the inferred invariants into higher-level observations and assertions. The work of Xie and Notkin [36] and Hankel and Diwan [10] appears to be a start in that direction.

Outcome filtering. Some outcomes reported by Agitator surprise users. For instance, Agitator might report that a method throws a `java.lang.NullPointerException`, because it was able to cause that exception by supplying a `null` value as a parameter. The users, however, might object to such a discovery, claiming that they *know* that a `null` value cannot propagate to that location in *their* system. At present, global data flow analysis that could detect this sit-

uation is beyond Agitator's capabilities. We are interested in finding mechanisms that would help Agitator in this, and other similar cases.

6.4 Studying the Testers

Experience suggests that having great tools is often insufficient to achieve wide acceptance of a new software-development practice. Developer testing is no exception. Unless the developers are motivated to make an effort (however small), they will not use even best of the tools. Unfortunately, very little is known about the human component of the developer testing practice and no conclusive evidence exists on what motivates developers to test (or not to test) their code. Myers provides some high-level insights in Chapter 2 of *The Art of Software Testing* [14], yet much more work is needed in that domain.

In his recent post to the Psychology of Programming mailing list [2], Ruven Brooks (one of the founding fathers of the field), poses the following questions about testing:

- How do people decide what to test?
- How do people construct tests?
- How do people actually use test tools (as opposed to the way the authors of those tools envisioned their use)?
- How do programming language syntax and semantics affect test strategy and behavior?
- How do novice programmers learn to do testing?

These questions have have no satisfactory answers in either testing or psychology literature.

7. CONCLUSION

Perhaps one of the most satisfying conclusions that we can draw from this paper is that academic research in software testing can and does have relevance to industrial applications. By paying attention to the usability, scalability, and performance requirements of commercial software development, we were able to deliver a tool that adapts a number of academic research ideas to the problem of developer testing.

The work on the Daikon invariant detector, as well as the wealth of research on test-input generation were instrumental to our success. Yet, we must ask why so few of the research ideas find their way into commercial applications. We do not have a definite answer to this question, but we suspect that the primary reason is the significant investment required to transform brilliant research ideas and prototypes into full-fledged, industrial-strength applications. The current version of Agitator (3.0) is the result of over 50 engineer-years of development and testing (with a very senior team of developers with significant prior experience in test automation and software development tools). And, as one might expect, the bulk of the development effort was not spent on the basic software agitation algorithms, but on usability, applicability, reliability, scalability, and performance. Tracking and addressing changing developers' preferences in terms of IDEs, frameworks, etc., is also a necessity for a commercial product. Even though support for a specific IDE is rarely a relevant factor in academic work, it is an imperative for a commercial product. For example, when Eclipse became the IDE of choice for most Java developers, we had to port the original version of Agitator from a stand-alone Swing application to an Eclipse plug-in.

Ultimately, however, if the problem solved is big enough—and software testing is definitely a big problem—we believe that the significant cost and effort required to productize promising research ideas is well worth it. Researchers get the satisfaction of seeing their ideas used broadly, software development tools companies are able to differentiate themselves through innovation, and users benefit by having more powerful and more effective products to help them deliver quality software.

We hope that other individuals and companies will follow suit in driving some of the best research ideas from recent years to commercial success.

8. REFERENCES

[1] K. Beck and E. Gamma. Test infected: Programmers love writing tests. In *Java Report*, volume 3, pages 37–50, 1998.

[2] R. E. Brooks. Commercial reality. Psychology of Programming Interest Group Mailing List, March 2005. `http://www.mail-archive.com/discuss@ppig.org/msg00958.html`.

[3] J. deRaeve and S. P. McCarron. Automated test generation technology. Technical report, X/Open Company Ltd., 1997. `http://adl.opengroup.org/documents/Archive/adl10rep.pdf`.

[4] Eclipse.org. Eclipse platform: technical overview, 2003. `http://eclipse.org/white-papers/eclipse-overview.pdf`.

[5] M. D. Ernst. *Dynamically discovering likely program invariants*. PhD thesis, University of Washington, 2000.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: 21st International Conference on Software Engineering*, pages 213–224, 1999.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[9] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE '03: 27th International Conference on Software Engineering*, pages 60–73, 2003.

[10] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP '03: European Conference on Object-Oriented Programming*, pages 431–456, 2003.

[11] Hybernate.org. Relational persistence for Java and .NET. `http://www.hibernate.org/`.

[12] D. Ingalls. Fabrik: A visual programming environment. In *OOPSLA '88: International*

Conference on Object-Oriented Programming, Systems, Languages, and Applications, volume 23, pages 176–190, Nov. 1988.
[13] JUnit. http://www.junit.org.
[14] G. J. Myers. *The Art of Software Testing*. Wiley - Interscience, New York, 1979.
[15] NUnit. http://www.nunit.org.
[16] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
[17] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP '05: European Conference on Object-Oriented Programming*, pages 504–527, 2005.
[18] S. G. Parker, D. M. Weinstein, and C. R. Johnson. The SCIRun computational steering software system. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*. Birkhauser Press, 1997.
[19] N. H. Petschenik. Building awareness of system testing issues. In *ICSE '85: 8th International Conference on Software Engineering*, pages 182–188, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.
[20] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference/11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, New York, NY, USA, 2003. ACM Press.
[21] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 478–488, New York, NY, USA, 2002. ACM Press.
[22] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 76–85, New York, NY, USA, 2004. ACM Press.
[23] S. Sankar and R. Hayes. Specifying and testing software components using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, April 1994. http://research.sun.com/techrep/1994/smli_tr-94-23.pdf.
[24] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: 10th European Software Engineering Conference/13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.
[25] SpringFramework.org. Spring application framework. http://www.springframework.org/.
[26] Sun Microsystems. Java platform, Enterprise Edition. http://java.sun.com/javaee/.
[27] The Apache Software Foundation. Apache Struts Project. http://struts.apache.org/.
[28] The Daikon Project. *Daikon Invariant Detector User Manual*, January 2006. http://pag.csail.mit.edu/daikon/download/doc/daikon.pdf.
[29] The Jakarta Project. Commons collections. http://jakarta.apache.org/commons/collections/.
[30] D. Thomas and A. Hunte. Mock objects. *IEEE Software*, 19(3):22–24, 2002.
[31] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/FSE-13: 10th European Software Engineering Conference/13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.
[32] R. Vanmegen and D. B. Meyerhoff. Costs and benefits of early defect detection—experiences from developing client-server and host applications. *Software Quality Journal*, 4(4):247–256, 1995.
[33] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, New York, NY, USA, 2004. ACM Press.
[34] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS '05: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 365–381. Springer-Verlag, Apr. 2005.
[35] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE '03: International Conference on Automated Software Engineering*, pages 40–48, 2003.
[36] T. Xie and D. Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *ISSRE '05: International Symposium on Software Reliability Engineering*, pages 277–287, 2005.