Abstract

We introduce a methodology and research tools for visual exploratory software analysis. VERDICTS combines exploratory testing, tracing, visualization, dynamic discovery and injection of requirements specifications into a live quick-feedback cycle, without recompilation or restart of the system under test. This supports discovery and verification of software dynamic behavior, software comprehension, testing, and locating the defect origin. At its core, VERDICTS allows dynamic evolution and testing of hypotheses about requirements and behavior, by using contracts as automated component verifiers.

We introduce Semantic Mutation Testing as an approach to evaluate concordance of automated verifiers and the functional specifications they represent with respect to existing implementation. Mutation testing has promise, but also has many known issues. In our tests, both black-box and white-box variants of our Semantic Mutation Testing approach performed better than traditional mutation testing as a measure of quality of automated verifiers.

VERDICTS:

Visual Exploratory Requirements Discovery and Injection for Comprehension and Testing of Software

By

Şefik Kanat Bolazar

B.S. Middle East Technical University, 1990

M.S. Syracuse University, 1993

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer and Information Science

in the Graduate School of Syracuse University

August 2013

This thesis is dedicated to my parents, Günay Bolazar and Bedriye Bolazar,

who have shaped me in so many ways that I continue to discover to this day...

Contents

List of Tables

List of Figures

Chapter 1

Partial Automation of Software Component Verification

1.1 Contributions

Our research contributions:

- Revealed some essential problems in traditonal debugging paradigm and traditional mutation testing.
- Developed a method for real time exploration of complex software, supported by formal contracts to record hypotheses.
- This work enables contracts to be specified and edited in real time with support for deep analysis of their consequences.
- Demonstrated how dynamic requirement and behavior contracts can be used for software analysis and testing in various scenarios.
- Provided means to evaluate the effectiveness of contracts using mutation testing in a way that, we believe, is superior to existing methods.

- Developed the VERDICTS tool to support the use of these explorations to better understand complex software.

VERDICTS tool combines a number of innovative features:

- Integrating large amounts of program values in comprehensive but useful ways
- Using novel visualizations that reveal patterns in control flow and data variations
- Seamlessly switching between modes of analysis and testing
- Dynamically inserting probes and hypotheses about program behavior using a familiar language
- Tracking assertion results in space of source code and time of execution

1.2 Partial Automation of Software Component Verification

"Everything should be as simple as possible, but no further." Albert Einstein (attributed[1]) (1879 - 1955)

1.2.1 Automated Verification of Undocumented Software Components

Recently popularized Opportunistic Software Systems Development paradigm [64] proposes reusing code that was not initially intended or designed for reuse. One source of reusable code is the large body of open source software, including much free software with license

[1] *"It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience."*, Albert Einstein (1879 - 1955), in "On the Method of Theoretical Physics" The Herbert Spencer Lecture, delivered at Oxford (10 June 1933); also published in Philosophy of Science, Vol. 1, No. 2 (April 1934), pp. 163-169, p. 165

that allows reuse in commercial software. Reuse is made easier by the open source software code search engines such as Koders [49] and Krugle [50].

Unfortunately, open source software that is verified in the field by a large number of end users may lack proper documentation or automated verification, making comprehension and reuse of any part of this software very difficult.

For complex software without proper documentation or automated verification, we propose a partial-automation approach that:

- Is designed for exploratory software and component verification
- Allows quick switching between software verification and analysis (for comprehension)
- Employs dynamically discovered component specifications for both verification and analysis
- Can evaluate the adequacy of discovered component specifications as they are evolved
- Can generate automated component testers based on discovered component specifications

Before we examine our proposed approach in more detail, let us examine potentially desired features of such an approach.

1.2.2 Software Development Stages, Cycles, and Feedback

Winston W. Royce's structured approach to software development conceived in 1970 recommended simple segregation of software development tasks and stages, followed once or twice through, without cycles [72]. Royce's software development approach was later named "the waterfall model".

Royce also recognized that a running system reveals all the side effects, feasibility and compatibility of desired features and design decisions. Even in 1970, Royce suggested that one could “do it twice”, allowing what is learned from interaction with the first developed system to inform all decisions in designing, implementing and testing the second system.

Today, agile software development methods [1] suggest rapid development and delivery cycles (iterations) as short as one week. Within one iteration, agile software shrinks cycles of feedback further with continuous integration and test automation (as well as on-site customer), where implement-integrate-verify-learn cycle with valuable feedback is often shrunk down to hours and even minutes.

Before we receive feedback from our observations, we do not know if the hypotheses behind our actions are correct. No learning takes place before our actions (such as implementation of a feature) produce observable feedback. Faster feedback allows more cycles of learning to be performed within a given time, which means that shorter delay in feedback makes learning more efficient. Shorter feedback delay also improves efficiency of control.

Consider again the issue of opportunistic software reuse. Software that is widely deployed may be verified in the field with a large number of end users, even when good software development practices are not followed. Such software and its components may have no automated verification mechanism, no requirements specification document or any other documentation.

1.2.3 Software Component Testing and Analysis

Much simple mechanical scripted testing can be off-loaded to automated unit tests, leaving test analyst free to perform testing tasks with higher conceptual complexity and critical thinking. Without a predetermined path, the space of possible test cases can be explored by individual decisions that allow perceived highest-remaining-risk test cases to be tested at each next step.

Exploratory testing [9] recognizes and leverages the fact that when not following a simple mechanical script, test analysts not only test, but also analyze; not only verify software, but also generate, verify and modify their own hypotheses about how the software should behave.

To achieve this, exploratory testing proposes rapidly switching between testing and analysis modes of operation to perform unscripted testing efficiently.

Of interest to software reuse is software component analysis and verification. How can we assist these two tasks, to partially automate an unscripted, exploratory approach to software component verification?

1.2.4 Software Comprehension Theories, Importance of Hypotheses

The end goal of analysis is software comprehension. Even though not an explicit stage in software development or maintenance, much time is spent on software comprehension.

After first deployed version, software must be maintained for the rest of its life, and this maintenance includes added features ("perfective maintenance") and adjustment to changed environment ("adaptive maintenance"), not only fixing bugs ("corrective maintenance").

Today, maintainers spend about 40% of time understanding software, and lack of documentation is one of the main reasons [46].

There are many competing and incompatible software comprehension theories. Short of statistical analysis of a large user base with subjective exit survey about level of comprehension, there is no generally satisfactory way to quantify or prove claims about comprehension. This has caused the field of software comprehension to shrink over the decades, and caused most research on software analysis to avoid making any claims about comprehension.

Case in point: PCODA, the International Workshop on Program Comprehension Through Dynamic Analysis has skipped years 2009, 2011 and 2012. Proceedings of PCODA 2010 [37] (available online at [68]) only has three papers, which mention comprehension only in Introduction, Background and Related Work sections and never in any other section of any paper. None of the papers refer to or suggest any comprehension model or theory, and no paper makes any claim about improved comprehension or describes any experiment with humans to assess in any way whether their technique improves comprehension.

Even though we had originally designed our approach with the end goal of improved software comprehension, we will also avoid making any claims about comprehension at this time. Still, we believe that it is important to learn from various software comprehension theories.

Even though various software comprehension theories propose conflicting structures and processes, practically all theories heavily emphasize the importance of hypotheses (also called "conjectures") and sub-hypotheses, their creation, evolution, verification (for an accept/reject decision), and possible abandonment.

Abandonment refers to unresolved hypotheses that are ignored, either because an accept/reject decision could not be made or due to lack of follow-up [59], possibly due to forgetfulness. The truth is, not all hypotheses need to be resolved; there may be many parallel paths to test or prove a primary hypothesis. Some hypotheses and many supporting sub-hypotheses may be non-essential and redundant, and need not all be resolved in order to make decisions on essential hypotheses. If verification tests are inefficient or inconclusive, an non-essential hypothesis may be abandoned in favor of other verifiable hypotheses.

Vessey [79] has observed that novice developers are more resistant to rejecting/changing their hypotheses. Novice developers are more likely to state hypotheses, assume them correct, and infer incorrect conclusions, whereas expert developers are more likely to accept the unpleasant choice of rejecting their own hypotheses in the presence of contrary evidence.

It has also been observed that expert programmers abandon hypotheses more frequently to look for other faster approaches to reach their goals [79] [59].

Just as software/component analysis can be conducted through initially unconfirmed hypotheses of intended (desired/expected) program behavior, testing can be conducted through hypotheses of actual program behavior (essentially, that the program behavior *is* as expected). A simple segregated view of these two tasks is shown in table 1.1.

In practice, debugging almost always, and testing often switches between these two modes of operation, and some experiments actually combine testing and analysis, with aspects of observed results verifying both types of hypotheses at the same time.

Table 1.1: Software testing, analysis, and hypotheses

Task	Trusted	Suspect	Hypotheses of	Failure Indicates
Test/Debug	Tester	Software	Actual Behavior	Bug in Software
Analyze	Software	Analyst	Intended Behavior (Requirements)	Faulty Mental Model

To assist analysis and testing tasks, we need mechanisms to allow such hypotheses to be:

- generated, modified,
- verified against the software,
- accepted or rejected, and at times, abandoned.

1.2.5 Declaring Hypotheses

Automated unit testing promises full automation once a representative sample set of test cases are gathered. Generally, the test cases are individual entities with memorized inputs and expected outputs or side effects. Even though we can also create an automated tester by combining a general-purpose verifier with randomly generated or sampled test inputs, this approach is practically never used because it is considered much more costly to generate such a general-purpose verifier for a component.

Full test automation has the goal of generating a simple scripted mechanical process. A human could also follow this process mechanically and mindlessly without knowing or learning anything about the component or its requirements. Our goal is instead to empower a test analyst or a developer with tools that help explore, analyze and verify software. With memorized test cases, an automated unit test suite:

- Does not explicitly declare hypotheses (predicates) of expected software or component behavior

- Is not generalizable, and therefore cannot handle externally generated component or program (test) inputs

In contrast, most specification languages allow specification of requirements that:

- Explicitly define hypotheses (predicates) of expected software behavior
- Are designed to handle externally generated test inputs

We are specifically interested in object-oriented specification languages which often define requirements for each class and its methods (member functions). Three commonly used types of OO requirements are:

- Method preconditions: What the method expects from its inputs and object state.
- Method postconditions: What the method promises (in outputs, modified object state, modified inputs)
- Class invariants: Object internal consistency requirements; what all methods promise to preserve on each execution, upon completion of execution.

It seems reasonable to use an object-oriented specification language as the primary means of recording and testing hypotheses of expected and actual behavior. But this language should be as familiar and as simple as possible for the developer; [10] reports that fourth-year review of ADL has revealed that even after a few years of usage, a well-designed, powerful but unfamiliar specification language does not necessarily improve productivity due to steep learning curve involved. What is more, as specifications grow, their maintenance can become extremely difficult [22], and keeping the specification language simpler could reduce the likelihood of harboring bugs in specifications.

1.3 Problem Scenarios

In the next few sections, we briefly introduce five problem scenarios:

1. Testing, Verification
2. Analysis and Comprehension
3. Debugging
4. Specifications Discovery
5. Measuring Automated Verifier Adequacy

These scenarios will be examined in more detail and analyzed along various dimensions in chapter 2, in sections 2.1.1 - 2.1.5.

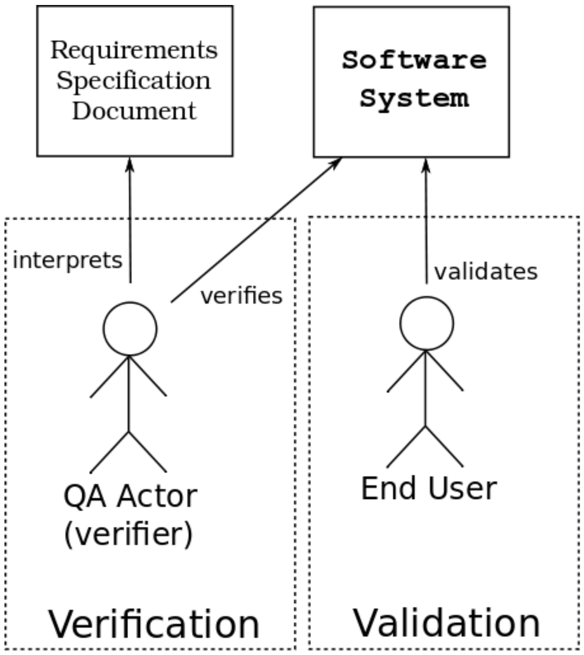


Figure 1.1: Software System Verification and Validation

1.3.1 Testing, Verification

Verification & Validation is the final software development stage [2] before software is deployed to end users. This stage really consists of two stages (see fig.1.1):

[2]In the test-driven development and agile methods, different from the waterfall and iterative methods, automatic unit testing is required and actually must be coded before program component being tested is implemented, so some verification effort appears before implementation effort. Still, there is a need for system verification and validation before deployment.

- Verification, with respect to specifications, agreed-upon documents, contracts
- Validation, with respect to end users' actual needs and desires

Verification used to be mainly conducted after the implementation stage by test analysts or other QA (Quality Assurance) personnel. Today, with the popularity of unit testing, prior developer testing that is conducted during implementation stage often greatly augments the verification conducted by test analysts. Throughout, the developer and the test analysts put themselves in the shoes of the end users, but their guesses to resolve ambiguous requirements still need to be validated by end users.

Software is a conceptually complex entity. Feasibility, compatibility, and side effects of a large set of requirements may not be apparent to end users from the start. Upon using the finished product, end user may realize some requirements could be dropped, others may need to be added or clarified. It is not as safe for developers to make these decisions, so we cannot skip the validation stage.

Expanding the reach and improving the quality and efficiency of verification can help reduce some of the complexity and costs of software validation. To automate verification, we need to use a machine-understandable specifications language. But a specifications language that is not familiar to end users, and that also cannot easily be interpreted by developers or test analysts for the benefit of the end users would make validation stage fraught with confusion. Therefore, when picking a specification language, a balance between familiarity, comprehensibility and automatability must be struck.

1.3.2 Analysis and Comprehension

Even though many analysis tools aim to improve comprehension, as we noted earlier, researchers rarely make claims, quantify, or attempt to prove improvement in comprehension. Because the human mind is highly complex, nonlinear and plastic, it is very difficult to develop a universally acceptable objective metric of comprehension.

There are a large number of competing and incompatible software comprehension theories. Some theories focus predominantly on a bottom-up traversal, others on a top-down traversal, and yet others state both these strategies are employed during a comprehension task.

Most software analysis tools that help developers use visualizations, feature location in code, and tracking program state (held in data) during execution. If we could combine these primary features of software analysis tools in a one-stop software analysis tool, we could:

- Reveal implementation-feature links through dynamic execution and white-box analysis of internals
- Reveal links between code (static) and data (dynamic, program state)
- Aid analysis with well-designed informative visualizations
- Support both top-down and bottom-up strategies

As mentioned before, one other feature enables not only analysis but also testing:

- Support for hypothesis generation, modification and verification, for actual and expected component behaviors.

1.3.3 Debugging

Presence of bugs (defects) in software are revealed when they cause failure/fault (inaccurate behavior; behavior that is inconsistent with requirements) that is observed by the user. Often, a period passes before internal state corruption reveals itself as observable faulty behavior, and user may not notice the failure for a while even after that point in time, as shown in fig.1.2.

When a critical bug is discovered in deployed software, the turnaround time of a fix for this bug reflects greatly on the perceived quality of the software and the company that produced the software.

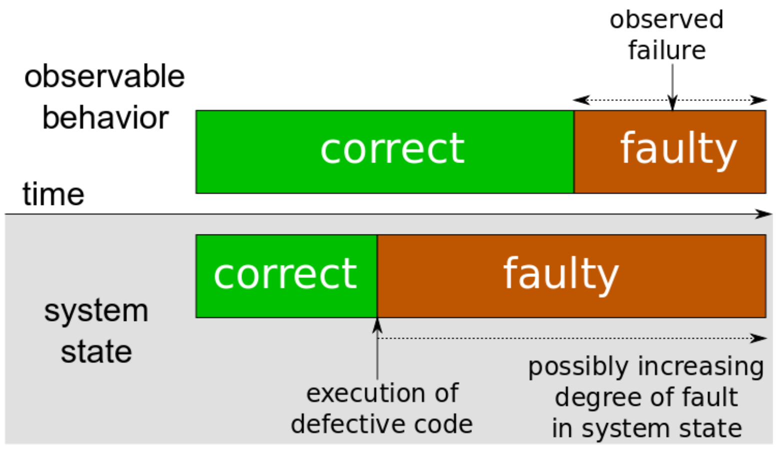

observed failure
observable behavior
correct
faulty
time
correct
faulty
system state
execution of defective code
possibly increasing degree of fault in system state

Figure 1.2: From defect (fault) in software, to faulty state, to observed failure (faulty behavior).

Debugging really has two distinct steps:

- find the bug (discover fault origin; “fault localization”)
- fix the bug

The first step, “fault localization”, can be partially automated. If the bug is trivial, the solution (step 2) may require a very small change in code. But not all bugs are caused by simple typos, so the second step, “fix the bug”, cannot, in general, be automated. Fixing the bug may require some redesign and reimplementation, which require creative input and problem-solving by the developer. For nontrivial bugs fixing the defect involves cycles of software development (implementation + testing).

The standard debugging paradigm attempts to support the first step, is 50 years old, and has not changed much since 1961 (see section 2.1.3). It is also not conducive to an exploratory mode of thinking; its basic mode of operation is a linear traversal of individual instructions executed over time. But today’s computers are more than a million times faster when compared along average C program execution speeds by using Dhrystone benchmark [41].

Due to this and various other reasons we will examine in the next chapter, debugging remains inefficient and is often reported to be a frustrating experience. In the next chapter, we will examine some innovative approaches to debugging, as well as why and how a standard debugger without memory of program state history may give us less useful information than time honored low-tech practice of using print statements.

Our analysis in large part agrees with conclusions of Agitator [18], an industrial automated developer testing tool that implemented some research ideas out in the field. A method/tool that helps with debugging should:

- Partially automate first step, discovering defect origin
- Use familiar and intuitively simple languages and forms of interaction
- Have wider focus than a standard debugger:
  - Help see dynamic context of execution & program state
  - Lead to exploratory rather than linear behavior
- Be efficient, and apply recent methods
  - Use efficient heuristics to speed up algorithms with high time complexity
  - Process software at certain points and times instead of at every instruction

1.3.4 Specifications Discovery

Both in preparing a component for reuse and in reverse engineering, an existing, field-tested component may have to be retrofitted with requirements specifications document and/or automated unit tests. Even when software has specifications, they may be faulty, old, or obsolete, especially when they are not directly used or referred to in the development cycle.

By using a readable but also machine-interpretable specifications language, we can combine automated verification with requirements specification discovery and documentation.

Ideally, such specification languages can be used to declare two types of hypotheses mentioned earlier:

- actual behavior hypotheses, for analysis, and ultimately, specifications discovery
- expected behavior hypotheses, for testing/verification (the tester knows what is expected)

Specifications discovery is part of reverse engineering, and goes in the reverse direction of implementation. During implementation, the requirement specifications are trusted and the implementation (and therefore, the component behavior) is generated and modified. Specifications discovery is the reverse process, where the implementation and the component behavior are trusted and the requirement specifications are generated and modified.

1.3.5 Measuring Automated Verifier Adequacy

For a component that does not have an automated verifier, specifications discovery may propose one or more verifier candidates. For a component that has a verifier, we may be interested in finding out if the verifier is up-to-date and aligned with the current behavior of the component.

One possible approach is to combine the verifier with a rich set of test inputs to create an automated tester (a test harness), and then to evaluate the adequacy of this automated tester.

Various code coverage criteria can be used to evaluate the adequacy of an automated tester. Unfortunately, code coverage criteria ignore the output of the component and pass/fail judgment of the verifier completely, and only measure test input set adequacy. Our focus is the rather orthogonal task of measuring verifier adequacy.

Mutation Testing (MT) can also be used to evaluate the adequacy of an automated tester by using multiple modified versions ("mutants") of code. MT depends on the pass/fail judgment of the verifier, and therefore can be used to evaluate verifier adequacy. But MT has a number of shortcomings mostly stemming from focusing on single syntactic difference mutants even though most such mutants generate large semantic/behavioral differences. We examine these shortcomings in some detail and propose an alternative in section 1.5.3 below.

1.4 Observations and Proposed Solution

Here are some of our observations so far:

- Software testing and analysis tasks may benefit from switching and mixing in real time
- Smaller cycles and faster feedback can improve efficiency of learning and control
- Accurate up-to-date requirements specification document would help with both analysis and testing of software
- Automated verifiers and tests can improve efficiency of some software development tasks
- Focused visualizations can aid analysis (both static structure and dynamic behavior analysis)

Even though we have mostly focused on testing and analysis tasks, we will see in chapter 2, through analysis of various scenarios along various dimensions that these observations are generalizable to many other software development tasks.

These observations directly correspond to the features of our proposed solution:

- Allow quickly switching between software testing and analysis modes of operation
- Allow mixing software testing and analysis with dual-purpose hypotheses
- Minimize delay between implementation/modification/declaration of specifications and execution within integrated system; avoid need for full system recompilation and restarting
- Record and gather discovered specifications as component verifiers

- Automate comparison of component verifiers using a verifier adequacy metric
- Use various visualizations to aid various static and dynamic analysis tasks
- Automatically generate test harnesses (automated software/component testers) by combining component verifiers with input data from system trace

These features are provided through our VERDICTS (Visual Exploratory Requirements Discovery and Injection for Comprehension and Testing of Software) approach and research implementation. Our implementation uses method preconditions and postconditions to record component specifications. We provide a relative adequacy metric for automated verifiers using our Semantic Mutation Testing (SMT) approach. The next two sections describe briefly how VERDICTS and SMT work.

1.5 Method Description

1.5.1 VERDICTS: Visual Exploratory Requirements Discovery and Injection for Comprehension and Testing of Software

VERDICTS is the name of both our new method and our research implementation for visual exploratory software analysis. VERDICTS combines exploratory testing, tracing, visualization, dynamic discovery and injection of requirements specifications into a live quick-feedback cycle, without recompilation or restart of the system under test. This supports discovery and verification of software dynamic behavior, software analysis, testing, and locating the defect origin. Our research implementation uses aspect-oriented method call interception, functional specification of requirements using dynamically injected method preconditions and postconditions, statistical analysis and various forms of visualization.

Specifically, our research implementation of VERDICTS combines:

- Program trace visualization, with capability to zoom in to method call details (inputs, outputs, "observables")

- Interactive declaration of observables (watch expressions that are logged) for the internal state of the system
- Statistical and visual analysis of the collected observable data
- Programmatic declaration of expected behavior of the system defined through invariant relationships and satisfaction of method preconditions and postconditions on these observables
- Recording of any GUI component, for synchronous playback with trace visualization
- The ability to modify and repeat these steps without restarting the system under test (SUT)

The rapid hypothesize-test cycle of VERDICTS provides feedback that is:

- Quick, allowing efficient operation and ability to quickly discard faulty mental models (conducive to efficient learning and improved control)
- High in information content
- In a form that makes information and patterns easy to grasp
- Helpful in discovering the software component responsible from a feature or a certain behavior

These features allow VERDICTS to provide a rich environment for dynamic (on-the-fly) creation and evolution of method specifications that can be tested without restarting the software under test. These method specifications can then collectively be used as a component verifier. Together with trace data collected during execution, VERDICTS can generate an automated test harness for these methods.

Instrumentation necessary to collect trace data is feasible; large amounts of data can be gathered without significant performance penalty while the visualizer remains responsive to tester interaction. Our personal experience is that the system is very quick to set up, faults are discovered quickly, and inefficient algorithms (which may produce correct results) become glaringly obvious through our method trace visualization. We will examine VERDICTS in full detail in chapter 4.

1.5.2 Dynamically Generated and Evolved Requirements Specifications (Contracts)

VERDICTS research implementation uses method preconditions and postconditions written in Java (rather than use a specification language). Even though many OO specification languages depend on pre- and postconditions, we consider our implementation to be programmatic Design by Contract (DBC) [60] in Java. Our methods can be applied to any specification language and any implementation language.

Bertrand Meyer's DBC is a lightweight, inlined object-oriented component specification approach. In DBC, classes declare contracts, specifying what each method (member function) expects (its precondition) and promises (its postcondition), while all methods, on exit, preserve the object internal consistency requirements (class invariants).

In the waterfall model, every component that is to be implemented must have requirements documented before implementation of component begins. Similarly, traditional DBC approach expects contracts to be written for each component, before implementation. Using DBC pervasively for all components can add significant implementation costs [62] Instead, targeted selective application of DBC may still yield significant improvements in software quality.

To combine exploratory testing and analysis, we suggest generating DBC contracts "in situ", by discovering expected behavior for components that are already implemented and integrated in a properly functioning software. Instead of full coverage, we propose a targeted application of contracts that allows prioritizing components according to task at hand.

Beyond contracts for hypotheses of expected behavior (for component testing), we also propose using contracts for hypotheses of actual behavior (for component analysis). This uniform interface allows not only quickly switching between but also mixing testing and analysis modes of operation.

1.5.3 Evaluating Requirements with Semantic Mutation Testing (SMT)

VERDICTS can generate automated component testers ("test harnesses") by combining evolved component verifiers with input data from system trace. The user can also add other inputs, possibly by using test input generators.

We are interested in analyzing field-tested software with VERDICTS. Even when software is not generated with good practices, it may be extensively field-tested and validated by end users, with discovered bugs fixed by developers. Such software would not be in wide use if it still had frequently manifested or critical bugs. In the testing-analysis spectrum, we will use VERDICTS initially and more often for analysis, assuming software behavior essentially correct, analysts' hypotheses to be suspect, when behavior does not match hypotheses.

A component integrated in such a system is also indirectly field-tested by end users. Within an integrated system, a component's inputs can be constrained to a small input domain, and not all outputs may be used. Still, for correct operation, the system demands certain behavioral requirements to be satisfied. Within the operating conditions of the component integrated in this system, the component must also behave correctly for the system to behave correctly. A component may have been intended to work correctly on a more general set of inputs, but as long as we are only interested in verifying this system rather than planning to reuse the component, we need not consider generalizability of the component beyond the needs of the system within which it is integrated.

Because we assume that the component's current implementation produces desired behavior, our verifiers must learn from the implementation in order to extract and declare the requirements that are implicit in the implementation.

An adequate verifier (contract) should check all aspects of the behavior of the component. In order to measure the degree of match, we need a way to evaluate concordance of the verifier (and the functional specification that it represents) with respect to existing implementation. We need to measure the concordance between the component specification/verifier and implementation.

One idea is to evaluate the automated tester that contains the verifier, using various test adequacy metrics. Unfortunately, as we mentioned before, code coverage criteria ignore verifier completely and only measure test input adequacy.

Mutation Testing [65] (MT, "traditional MT") subsumes code coverage criteria such as statement coverage, branch coverage, and multiple condition coverage [66]. MT also depends on the pass/fail judgment of the oracle/verifier, so it may potentially be used to measure the adequacy of test oracles (and the functional specifications they represent). Various researchers report a number of problems with MT:

- MT has high computational complexity, mostly due to compilation of each mutant program.
- Checking for semantic equivalence of mutants as compared to the original program is a tedious, manual task [36]
- Many mutants are so different in behavior that they may fail almost all tests ([44] had about 41% of all mutants fail all tests, and called these "dumb mutants")
- "Competent programmers write programs that have few defects" does not mean competently written programs have small syntactic differences (Jia and Harman [44] call this misconception "Syntactic-Semantic Size Myth")

For a more detailed critique of traditional Mutation Testing, see section 5.4.

In order to answer these problems, we suggest creating mutants dynamically (without recompilation) and focusing on generating and controlling semantic defects, by measuring and quantifying differences in program state and component behavior.

We developed Semantic Mutation Testing (SMT) as an approach to evaluating concordance of automated test oracles (verifiers) and the functional specifications they represent with respect to existing implementation. In the analysis of our small-scale tests, we compared black-box and white-box variants of SMT against traditional MT and found SMT to produce a better measure of the quality of automated test oracles [14].

For a detailed description of Semantic Mutation Testing, see section 5.5.

Chapter 2

Problem Scenarios

2.1 Problem Scenarios

Before presenting our proposed solution, we will examine some software development scenarios that are of interest to us:

- Testing, Verification
- Analysis and Comprehension
- Debugging, Fault Localization ("Find the Bug")
- Specifications Discovery
- Measuring Automated Verifier Adequacy

These scenarios are sorted essentially in order of popular interest in the research field of software development. Scenarios of popular interest often happen to be common scenarios for an average software developer as well, as common problems are more widely studied. An average software developer may more frequently encounter and work more towards scenarios listed earlier as compared to those listed later.

Some but not all scenarios also correspond to waterfall software development stages, even though in today's development, stages are not isolated. Continuous testing paradigm, for example, proposes continuously cycling between coding/development and testing stages. The last two scenarios are not necessarily common scenarios encountered by all software developers; they are instead specific scenarios that define our main area of research.

Scenarios of popular interest can be considered to be more important to the field of software development. For example, testing is important because of its significance in improving overall quality of the software. When critical bugs appear after deployment, debugging becomes important due to the sense of urgency of the problem, but in terms of time spent, debugging is a poor way to improve the overall quality of the software. Researchers focus on and promote better software practices, so there is much more focus on improved and alternative testing methods and relatively little focus on improved or alternative approaches to debugging.

This order need not represent the order of critical importance of various problems for any one software product at any one time. For example, when deployed software is discovered to have a critical bug, debugging becomes the scenario of focus. In the long term, we need to remember that even though there is no way to eliminate all bugs, we can reduce the frequency of such incidents by discovering more bugs before deployment if we employ better testing practices.

2.1.1 Testing, Verification

As stated in chapter 1:

- Verification can be automated by using machine-understandable specifications language.
- Validation cannot be automated or skipped.
- Validation can be made shorter by expanding the reach and improving the quality of verification.

- Validation may be made easier by using a human-understandable specifications language.

The contract between the software producer and consumer is the requirement specification document. Developers and test analysts interpret the specification document to conduct verification tasks that are aligned with the end users' needs. Specification documents are often written in unstructured or structured natural language statements, so they require human interpretation.

Some researchers promote formal methods as an approach for full automation of software verification. But most developers and end users are not familiar with formal methods, and find them difficult to comprehend. When formal methods are employed, the specifications and correctness proofs themselves may contain bugs that cannot be easily discovered. Formal specifications often use cross-cutting requirements specifications that cause non-local dependencies which make specifications difficult to maintain and modify [22]. To improve our confidence, automatic verification should be implemented in a language that is not overly complex or unfamiliar to the developers and/or test analysts.

The ideal automation we can hope to achieve would then require:

- Creating automated verifiers
- Using a specification language that is:
  - machine-understandable
  - understandable by (possibly already familiar to) developers and/or test analysts

In this document, for a given program or a component, we will use the term "verifier" [1] to mean an automated system capable of verifying whether a given set of inputs and

[1] N.B. This author believes that the common software testing term "passive oracle" is a misnomer, and "verifier" is a much more accurate and descriptive term. Oracles predict. Oracles answer questions. Creating an answer is almost always computationally much more complex than verifying the answer. What is more, researchers rarely make a distinction between passive and active "oracles", and "oracle" is used widely to mean "passive oracle". "Passive oracle" suggests, and "oracle" directly claims to have the complexity of an Oracle

outputs represent correct operation. Note that a verifier need not be a predictor capable of producing correct outputs. There may even be nondeterministic specifications that allow multiple sets of correct outputs. For example, the heap property used in heapsort could be satisfied in various ways for a given array of input values, and all that we need for a heapsort implementation to work is that heap property is preserved throughout once the heap is constructed.

2.1.2 Analysis and Comprehension

As stated in chapter 1:

- There are a large number of competing and incompatible software comprehension theories.
- Practically all major software comprehension theories emphasize the importance of hypotheses (hypothesis formation, modification, verification, and at times, abandonment).
- Human mind is highly complex, nonlinear and plastic, making it very difficult to develop a universally acceptable objective metric of comprehension.

Earlier software comprehension theories from 1970s and 1980s [74] [20] [69] were called program understanding theories. These studies borrowed heavily from natural language comprehension theories, they exhibited a heavy discourse focus, emphasized static code analysis (reading the source code and documentation), and often completely ignored dynamic analysis (running, observing, interacting with, and testing the software). This preference is probably in part due to lacking an end-to-end compileable and running program during many

without actually containing such complexity. Therefore, these terms are inflated and misleading compared to the straightforward and clear functional term "verifier". If the verifier actively and separately calculates the expected values in parallel (using a different algorithm and implementation), only then can it really be called an oracle, or simply, a predictor.

stages of software development before the advent and practicality of continuous software integration.

Most early comprehension models can be classified into top-down and bottom-up models [59]. Some models [74] [57] [59] are combined or opportunistic, where the developer is theorized to switch between bottom-up and top-down strategies as needed. For further analysis of software comprehension theories, see appendix A.

For many practical reasons, comprehension/recall of software cannot be complete with today's large-scale software, because:

- Software is too large to comprehend in full in one sitting; actually, it may take years to read the source code once through.
- Software (unlike a book) does not have a linear narrative; there might even be cyclic dependencies [31].
- Our memory (recall) is imperfect.
- Complexity and recall problems also apply to:
  - the domain (the vertical) of the application
  - the tools used, including programming languages and libraries used
  - the environment, including other executables, OS, and possibly hardware.

For a more detailed analysis, see appendix B.2.

In part due to these, and possibly due to the sensitive nature of evaluating individual software developers' skills and knowledge, industry also ignores comprehension and does not attempt to measure it during software development. For a more detailed discussion of essential incompleteness of comprehension and how it is often ignored, see appendix B.

As software is too large to understand with full detail, visualizations that can reveal high-level patterns can be of great help to comprehension. Such informative visualizations can reveal:

- Hierarchies and structure, of functions, data, and call graphs
- Relationships between software's static (code) and dynamic (data, program state, and behavior) aspects
- Code-code relationships through shared data (definition-use pairs, slicing, OOP encapsulation)

2.1.3 Debugging, Fault Localization ("Find the Bug")

In the next few sections, we will examine the standard debugging paradigm and alternatives, to attempt to answer questions such as:

- Why do software researchers often ignore debugging?
- Can we automate fault localization?
- Why does the standard debugging paradigm often cause much frustration?
- Is there something inherent in the 50-year-old standard debugging paradigm that causes a sense of lack of comprehension and control of software?
- What other common methods can we use to discover program behavior?

Later, in section 2.3, we will further analyze standard and innovative debugging techniques as we work through a simple buggy quicksort example.

Debugging: Ignored in Software Engineering, Inefficient, and Costly

Debugging is, at least in appearance, the least scientific part of software development. Both the term and the study of debugging are often shunned by software developers and researchers. "Debugging" does not exist in standard stages of software development except as the much more positive sounding "corrective maintenance". Unfortunately this term does not help us understand at what stage of development or by what means debugging should be performed, because maintenance stage contains within itself cycles of development that go through almost all earlier stages of software development.

Why do researchers often avoid using the terms “bug” and “debugging”? “Bug” suggests that our software has ingested a foreign unwelcome autonomous agent of disease. Naturally, this can elicit disgust. Presence of “bug” suggests imperfection and incompetence, and can instill distrust of any software development processes as well as any QA/inspection processes employed. For any nontrivial software, no QA process (specifically, no amount of testing) can promise to eliminate all bugs. Even in cases where formal methods can be used to automate software construction, and prove software correct with respect to formal specifications, formal specifications themselves may contain bugs [22].

Is debugging really important? Not always. If deployed software does not happen to contain any critical bugs, there may be no urgency, and standard processes and release cycles may instead be used, possibly replacing debugging with code review or rewrite of a component. Unfortunately, we can never be certain that deployed software is free of critical bugs. When a critical bug (including any security vulnerability) is discovered, debugging takes on a great urgency and importance. The turnaround time to fix a bug can reflect on the perceived quality of the software and on the trustworthiness of the company that produced the software.

Existence of bug reveals lack of control and comprehension of software behavior. Original developer of software components may start a debugging session by assuming that most of the software is comprehensible and controllable, except for a small portion which contains one bug. Unfortunately, this often proves too optimistic and leads to much frustration as debugging often reveals further lack of control and comprehension of parts of software, their behavior, and their interconnectedness.

Very often debugging is performed by a developer who did not develop the software. In this case, a high-level comprehension of only the relevant parts is hoped to be operationally sufficient for the debugging task, and frustration may be caused by discovering counterintuitive behavior and connections between parts of software that require wider or deeper comprehension of software.

Fault Localization (Discovering Defect Origin)

As stated in chapter 1, debugging consists of two distinct steps:

1. Find the defect origin (by tracing execution backwards from discovered observed fault)
2. Fix the defect, so that the software exhibits desired (correct, proper) behavior

As mentioned before, the second step of debugging, fixing the bug, cannot in general be automated, as it may require creative input for a new design and implementation. Our main focus is the first step, fault localization (discovering defect origin). Practically all debugging tools and techniques are focused on locating the bug and discovering how the bug corrupts the program state.

Fault localization (and therefore debugging) is not a manageable or predictable process that can promise to deliver results within a reasonably well-known narrow window of time and work. Some bugs are extremely difficult to track down from observed failure to defect origin; it may take several developers many months to replicate the conditions and discover fault origin.

Consider the 2003 North American Blackout at GE Energy that cut off electricity to 50 million people in eight states and Canada. This was caused by a race condition in about four million lines of C code that had been online and appeared to be bug free for many years [70]. Discovering the fault origin took six employees eight weeks; a cost of one person-year of work just to find one bug.

In a paper examining a company's software problem reporting and resolution process, Mira Kajko-Mattsson et al. report that merely replicating a buggy situation in lab conditions may take several months of hard detective work [45].

What causes replicating the conditions and discovering fault origin to take a long time? Some reasons reported in [70] and [45] are:

- There is not enough information about what types of inputs (and consequent program state) cause the bug to be manifested (insufficient logs, inappropriate bug reports).
- There is too much program state data that need to be entered or set up.
- Software runs too quickly to observe, may need to be slowed down.

Instead of running the software directly, we could use a debugger, but standard debugger does not automate setting up the program state, and would run too slowly, possibly causing program to run differently.

Not surprisingly, developers often consider debugging (and specifically, fault localization) to be a very frustrating task.

Sometimes fault arises from miscommunication due to different expectations of interacting components. If the rest of the software does not prescribe either version of expectations on either component we have incomplete/ambiguous implicit and explicit specifications, and defect can be ascribed to either component, and fixed on either side. The choice modifies the interface and contracts between components (cross-component expectations). In such a situation, even the “location of defect” need not be unique.

Fault localization may be partially automated using various approaches that systematically reduce the amount of code that needs to be reviewed to find defect origin, such as algorithmic debugging [73], slicing [11], and causal chain delta debugging [23]. These approaches assist a maintenance person by reducing the number of lines of code to be examined, rather than fully automate the discovery of fault origin for a given manifested faulty behavior. Before we examine these and other innovative debugging approaches in more detail, let us take a closer look at the much more widely used standard debugging paradigm.

Standard Debugging Paradigm, 1961-2013

Day-to-day software development and evolution can be performed somewhat linearly and locally within one module or a class. In terms of nature of work and strategies used, testing and debugging differ significantly from this low-level, linear, local and constructive implementation work. Testing and debugging are instead nonlinear, exploratory and destructive/contrarian. This requires a change from developer's linear mode of thinking. We need to break and observe before we can fix and improve. Its open-ended exploratory nature makes debugging more an art than a disciplined methodical engineering practice.

What happens when a linear approach is brought into debugging? The developer, thinking linearly may create a tool that linearly combs through every statement ever executed during the lifetime of a program, putting each instruction under the microscope in order to find that one instruction that should not have been run. This approach sounds reasonable if you are thinking linearly, outrageous if you are in exploration mode of thinking. As we will see in section section 2.3, this approach can quickly become excessively expensive.

Even though there has been research on various innovative debugging techniques over the decades, the standard paradigm and techniques of debugging follow this extremely costly linear combing idea, and have not changed much in the last 50 years.

Bil Lewis in his omniscient debugging paper published ten years ago (in 2003) [58] had remarked:

> Over the past forty years there has been little change in way commercial program debuggers work. In 1961 a debugger called "DDT" existed on Digital machines which allowed the programmer to examine, deposit, set break points, set trace points, single step, etc. In 2003 the primary debuggers for Java allows one to perform the identical functions with greater ease, but little more.

Such debuggers demand a microscopic focus on a single executable statement in a single line of code and at a single point in execution time, revealing everything on demand for a single program state. Nothing else (such as any previous value of a variable) is stored, so everything else has to be kept in the mind of the developer while the developer takes microscopic steps forward.

Execution can only go forward, and as earlier program state is not stored, we cannot go backwards from observed failure to defect origin by traversing causal links backwards. Following causal links backwards often requires multiple runs of the debugger with same exact set-up. Fear of missing the fault origin event can cause developer to use microscopic steps. In 1961, DDT debugger ran on DEC machines such as PDP-3, which according to its specifications averaged 100,000 instructions per second [71]. Today, we still use the same stepwise method, even though one second of program execution traverses billions of instructions that have to be stepped through.

How many more instructions are run per second on today's systems compared to 50 years ago? CPUs do much more work per clock cycle compared to 50 years ago, so we cannot just compare clock cycles. We can compare the speed of running a C program that is designed to have experimentally discovered average proportions of C program elements for programs that do not do any floating point operations. This is exactly what Dhrystone benchmark does. Table 2.1 shows 40-years of manufacturer's specifications and benchmark test results (1971 to 2011) for some Intel CPUs [41]. The Intel 4004 microprocessor's speed at 92,000 instructions per second in 1971 is very close to PDP-3's speed of 100,000 in 1961.

Today, an average software developer may not be able to examine and understand a line of C code much faster than 40/50 years ago, but these numbers show that running C code has gotten about 1.4 million times faster in the forty years between 1971-2011, so there is that much more code to cover for one second of program execution. Compared to 100 x 10^3 instructions per second reported for PDP-3 used in 1961, Intel i7 2600K runs 128.3 x 10^9 Dhrystone instructions per second in 2011.

In section 2.3 below, we demonstrate on a small example the problems and inefficiencies of standard debugging paradigm.

Table 2.1: Forty years of Intel CPU speed (1971-2011) as specified by manufacturers or measured by Dhrystone benchmark. Table also shows that today's CPUs can do much in each clock cycle. Dhrystone MIPS (million instructions per second) is standardized with respect to IBM/370 from 1972, considered generally to run at about 1 MIPS with clock speed of 8.69 MHz.

Intel CPU	Year	Clock (MHz)	Dhrystone MIPS	instructions/cycle
4004	1971	0.74	0.092	0.1
286	1982	12.5	2.66	0.2
486DX2	1992	66	54	0.8
Pentium III	1999	600	2,054	3.4
i7 2600K (quad-core)	2011	3,400	128,300	37.7

Print Statements, Logging and Tracing

Even when a good IDE with debugging capability is available, a short "print" statement that is inserted in the right place can reveal much more about the fault that has caused an observed defect. A print statement can only reveal a small slice of the whole program state, so the what is printed must be carefully chosen. Also, for compiled binaries, this method requires recompilation. There are a few reasons print statements can be easier to work with compared to debuggers:

- They reveal the history and change in some part of program state.
- Repetitive labor of entering an expression for evaluation or storing the result displayed (as debuggers do not store history) is avoided.
- Developer need not use microscopic steps through the program; it can be just run to completion.
- When the print statements collectively reveal the execution context, the developer need not mentally hold the program code space and execution time context.

If the software already uses logging, logs could also be examined to aid in debugging.

Compared to print statements that focus on discovering reasons behind one defect, logging focuses on revealing overall internal state of the software and/or report potentially significant events during software's execution. Logging requires predetermination of level of detail needed in printing (logging) the program state. As logging is always on, level of information customarily used in debugging is not practical in logging due to speed penalty on the program and disk space overhead.

Tracing specifically tracks methods/functions called during program execution, usually for discovering the execution time complexity of parts of a program. Unlike print statements and logging, tracing can be used on any program without recompilation, as long as debugging symbols were not stripped off from the executable. Tracing can be expensive so it cannot always be on. Main purpose of tracing is to find time spent on each function. Some tracing tools also reveal memory usage to help discover largest objects, heap usage, or memory leaks. A by-product of tracing the function calls is discovering the call hierarchy during one execution. Note that in any one execution, some branches may never be taken and some functions may never be called.

Unfortunately, as tracing ignores data (messages, variable values, function arguments, etc), it does not reveal anything about program state, and is of comparatively little value to the task of debugging.

In section 2.3 below, we will compare standard debugging paradigm to print statements through a simple example, and examine some innovative debugging techniques.

2.1.4 Specifications Discovery

Specifications Discovery: Part of Reverse Engineering

Standard process of software development depends on preparing detailed, accurate specifications in the form of requirements specification document before implementation of features begins. The specifications direct design, implementation and testing. When such specifications are missing, implementation and testing tend towards informally stated targets or developer's own targets.

In standard stages of software development, specifications discovery falls under reverse engineering. Explicitly stated specifications are a record of knowledge/comprehension for a piece of software, and can be used to communicate knowledge of software. Developers who are working on their own well-understood code may not need to do much reverse engineering. Developers maintaining or regularly reusing other developers' code (or their own partially-forgotten code) may need to do some reverse engineering every day.

Documented specifications are central to software verification and validation, and are of great importance to the tasks of software analysis/comprehension, reuse, and various maintenance tasks (such as fixing, improving, and porting software). When specifications are missing and multiple developers (present or in the future) may need to work on these tasks, discovery of specifications can be worth the effort.

Developers interested in maintenance or reuse are not only interested in high-level specifications, but also in component/unit specifications, contracts, and assertions.

Software That Works For End Users, But Not For Developers

Our interest in this scenario started with our frustration while working on Azureus (now Azureus/Vuze [8]) code base in 2006. Azureus is a peer-to-peer file sharing client (peer) distributed under GPL license. By February 20, 2013, Azureus/Vuze, at 538 million downloads is #3 most downloaded project of all time at SourceForge [76]. Since version 3.0 in 2006, it has been distributed with Vuze [81], a restrictive-license polished consumer interface designed to push commercial content. Our focus instead has been on the content sharing engine, Azureus, including its user interface. Even though very popular and verified through end-user field testing, Azureus has no requirements specification document and no automated testing.

Azureus is proof that software can be feature rich, extremely popular, complex, actively maintained and developed, and yet without any documentation, tests, in-code comments that could help newcomers understand existing code. Azureus handles a large number of parallel network connections and open files that are filled in asynchronously at multiple spots at once. It also uses innovative live animated interfaces to show state of files and communication with network connections to peers (see at [81]). Azureus shows how a piece of software can work wonderfully for end users, and not at all for developers who are trying to understand, improve, fix or reuse components. Each piece of source code in Azureus appears to be owned and well understood by only one developer who works on it. Azureus is actively maintained; it is not legacy code - not yet.

This is actually a very common open-source software scenario, especially for software produced by a small team:

- Software already satisfies the end users; its behavior is satisfactory, and is generally free of bugs.
- Software has almost no external documentation; there's no specifications or design document, and no user manual.
- New developers therefore do not know the original requirements of the software and the various problems it attempts to solve.
- Source code has almost no comments; there's no interface, package, file, object, function description or in-function comments.
- There are no automated unit tests, no integration tests, no system-wide tests.

Obviously, "generally free of bugs" does not mean the software does not have bugs. It may actually contain some known noncritical bugs. It may also have a number of undiscovered noncritical and critical bugs that do not manifest themselves in common patterns of usage.

Sometimes this situation is further complicated by other factors that are less frequent:

- We do not have access to current end users' discovered knowledge of the software purpose and behavior.

- Code is very hard to follow due to use of cryptic names and inconsistent naming.

If the gained comprehension of software should be shared between multiple developers, it would make sense to create documents that describe what was discovered about the software. Here are some common types of documents we could aim to produce in this situation:

- A preliminary user manual, or some basic usage instructions, possibly with some use cases
- Architecture/structure and design documents or notes
- Documentation through direct change to code, to add comments, change naming of functions, variables, etc to improve code comprehensibility
- End-user specifications: Expected/desired (deduced from observed) behaviors and features of software as a whole
- Detailed specifications (for developers): Expected/desired behaviors of units/modules within software

Why Are Comments, Documentation, Specifications and Automated Tests Missing in Many Open Source Software?

The fact is any work on developer/unit testing and documentation takes time away from implementation. Many developers who enjoy building software learn individually, starting with small programs. They may not enjoy writing tests or documentation. Automated testing is often more palatable as it is also programming, even though the exploratory and destructive essence of testing remains, and this may not be understood, followed or enjoyed by developers.

Most open-source software is built by individuals with varying levels of skills, styles, communication skills and difficulties. Most open-source software does not make explicit the overall design and architecture, the list of requirements. Source code style wars between strict factions and ignorant developers may often be resolved by not requiring a uniform style or even a naming scheme.

Many good software development practices require inspections, and critique of code developed to make sure standards and practices are followed. One social reason for not following these practices is to avoid discouraging the developers who are very often volunteers that may very easily leave the project.

Although common, this is not at all unavoidable in open source development. In fact, the level of testing and quality of some widely used open source software can be better than that developed professionally for other developers by the most prominent software companies. Good practices and inspections of code by a few "committers" may discourage some volunteers, but can also instill a higher degree of respect and trust towards the project and the developers involved.

For example, consider the open source Jakarta Commons collections library. As of version 3.1, the unit tests included for this library achieve 77.7% code coverage by having test code to application code ratio of 1.62:1 [18] (measured in KLOC). In this case, test code is 62% larger than application code.

In [67], Pacheco has automatically created tens of thousands of "directed random test" cases per library to find errors in a few classes from Java JDK, Jakarta Commons, and .NET Framework, and discovered fewer errors per KLOC (0 - 0.09) in classes of Jakarta Commons library compared to other commercially generated and distributed libraries tested.

Communicating Comprehension: From Mental Models to Documentation

Software comprehension is hard to study in part because it is internalized, personalized knowledge by one individual. One developer's comprehension does not help a team when that individual is not available. In what formal document can we hope to externalize comprehension of software?

This question is researched as part of reverse engineering, and among other types of documentation, requirements specification document is also suggested as a format for communication of software comprehension.

It is easy to see that software comprehension is vital not only for reverse engineering, but for any type of software development activity. Even for software which does not lack high-level design specifications, it can be helpful to incrementally discover and document low-level implementation specifications. The idea is that such documentation can be generated by a developer after comprehension, and used by same and other developers later to aid in communication as well as testing.

Implementation specifications are often defined through assertions, invariants, preconditions and postconditions. Instead of analyzing the whole software, a task-oriented analysis may document components involved and interactions observed for one use case. For software without a requirements specification document, this idea suggests opportunistic incremental discovery of specifications by prioritizing source code according to importance of use cases or exceptional/faulty behaviors observed.

Discovering Specifications for Test Automation

The goal of discovering end user and developer specifications depends on the target audience:

- Human-readable specifications document can help both end users and developers understand the software, but can be ambiguous, and cannot directly be used in automated software verification.
- Machine-understandable specifications can allow automation of software verification, but can be harder to follow or maintain [22].

Note that these two goals are not mutually exclusive. In our research, we are primarily interested in test automation, so we focus primarily on the second goal, but also try to satisfy the first goal as much as possible: We prefer to keep the machine-understandable language simple and familiar to developers instead of using formal specification languages, because a document in a familiar language can help developers understand the software better as well.

Considered as a comprehension communication tool, a specifications discovery tool should allow developer to generate, test, and ultimately accept or discard specification hypotheses. Accepted hypotheses become predicates of requirement specifications.

If the specification language is machine-understandable, the discovered set of requirement specifications constitutes an automated verifier for the component in question. Note that this verifier verifies correct operation for one execution of the component, so a “pass” means component performs satisfactorily for one test case’s inputs; it does not mean we know component to be implemented correctly, and therefore to perform correctly for all inputs.

2.1.5 Measuring Automated Verifier Adequacy

Relative and Absolute Adequacy Metrics

Consider Azureus mentioned before, which shares many features with legacy code. Azureus has no requirements specification document and no automated testing. It must have been manually tested. It has been widely deployed, and therefore also stress tested in the field. It handles complex multithreaded operations and communication. On the whole, it appears to be implemented correctly; it has no critical bugs. Azureus does not have verifiers for its components. If a verifier candidate were to be proposed for a component of Azureus, how could we judge the adequacy of such a verifier?

Much like test suite adequacy criteria, we need a way to measure adequacy of verifiers, in order to compare them, and in order to judge whether one is sufficiently adequate.

For a verifier candidate, there may be two types of metrics that could be used to measure the adequacy:

1. Relative adequacy as compared against an alternative verifier

2. Absolute adequacy, to compare against another verifier for another program. With sufficient study, we may also discover a “practically sufficient adequacy score cutoff”,

that finds a good compromise between completeness and time spent.

The first type of metric defines a partial order between verifier candidates for one component, but does not produce an absolute measure that allows comparison between verifiers for different components: With relative adequacy, we cannot compare degree of adequacy of a verifier for one component against a verifier for another component.

Most test suite adequacy criteria use absolute measures, usually with 0.0 representing complete inadequacy, and 1.0 representing complete adequacy.

Baseline desired features for a verifier adequacy measure would be that it fully uses the range 0.0 - 1.0, and gives better scores for better, more strict verifiers:

- Worst, empty verifier should get 0.0, and no other verifier should get 0.0
- Best, maximally strict verifier for a program should get 1.0, and nothing less should get 1.0 (but there may be different syntactic decompositions with same behavior)
- If a verifier V_1 is not worse than V_2 it shouldn't get a lower score.
- Ideally, verifier can help discriminate: If V_1 is strictly better than V_2, it should get a higher score.

Later, in section 5.2, we will formally define a subsumtion relationship that will help us evaluate our own verifier adequacy metric in our experiments, based on these baseline desired features.

2.2 Problem Analysis

2.2.1 Actors: User and Developer's Mental Models vs. Software Behavior and Implementation

End User's and Developer's Mental Models and Expectations

End user is mainly interested in overall black-box behavior of software whereas software developer is also interested in the implementation details. The end user only needs to create a mental model of how the software should behave.

The software developer needs to create a more involved mental model, that encompasses not only how the software should behave (to the best of his/her knowledge), but also how it is implemented.

Thus, the end user's mental model, restricted to this software's requirements, is structurally simpler than that of the developer as it does not include the internal complexity of the source code.

If the end user and developer agree completely on what is expected of the software, the developer's mental model subsumes (can deduce everything about) the end user's mental model. But it is not realistic to assume that the end user and the developer agree to this degree.

Even though at times the developer acts as an end user to evaluate the software behavior, in the final analysis, the developer is not the end user. As the developer may have expectations that do not completely overlap with those of the end user, the developer may fail to notice a faulty behavior that the end user could have noticed easily.

The requirements specification document is the contract of understanding between the end user and the developer. Contractually, the developer can get away in any development cycle with satisfying only what is documented. Unfortunately, specifications document is often vague and incomplete, and it may even prescribe undesired or infeasible behaviors.

Between development cycles, the end user may have a chance to negotiate a better contract by modifying the requirements specification document to represent more closely the end user's mental model, possibly after modifying his/her mental model to account for what is learned about feasibility and cost of features.

Accurate up-to-date mental model of software is essential for a developer in these scenarios:

- Testing, Verification: Faulty model could ignore bugs, or mistake correct behavior as a bug
- Analysis and Comprehension: Comprehension may start with a mental model that is inaccurate, or may even start without a mental model, but should yield an accurate model in the end. Note that our mental models cannot completely constrain any software of substantial size; our mental models are bound to be incomplete, and therefore, at times inaccurate (see appendix B.2).
- Debugging: Bugs are often caused by errors in comprehension/mental model or errors in deduction and prediction of consequences of interconnected decisions made during development. Debugging allows incremental localized improvements in mental model of developer, and often, (but not always) incremental local changes in code.
- Specifications Discovery: Similar to comprehension, except, this should produce a document.
- Measuring Automated Verifier Adequacy: Without a mental model of what is expected of software and how its components behave, a developer could not inspect an automated verifier and be able to judge its completeness and accuracy. An accurate model allows the developer to notice when there are errors in software's and its components' behavior. By abstracting this knowledge and comparing this ability to judge against how the automated verifier judges the software allows the developer with an accurate mental model decide how adequate the automated verifier is.

2.2.2 Process: Stages, Cycles, and Feedback

Software Development Process, Stages of Development

In 1970, Winston W. Royce described his structured approach to software development [72], in part to address the common problems of budget and schedule overruns, damage to property and other risks (and later, damage to life, in Therac-25 incident, 1985-1987). Royce's model was later named the waterfall model, and its clear separation of stages of software development is conceptually easier to follow compared to other approaches.

Most recent approaches to software development observe the synergy between stages of development and recommend going through cycles, to allow later stages of earlier cycles to help make better decisions in earlier stages of later cycles. For example, feedback from implementation in the first cycle can help improve design in the second cycle.

Although Royce's model worked best with few and small cycles that do not go beyond supplying feedback to one earlier stage, even in 1970, Royce observed that there are cases of dependencies that go farther than one earlier stage, and he even suggested "do it twice": Go through the whole cycle twice, and use what is learned from usage of the first developed system to inform all decisions in designing, implementing and testing the second system.

Shrinking Software Development Cycles

Iterative software development (used as early as 1957 according to Gerald M. Weinberg [52]) and prototyping also allow what is learned from existing, running, feature-incomplete system to inform decisions made to design and build the full system.

Agile software development [1] uses development cycles (iterations) as short as one week.

Actually, design-implement-test cycles in Agile Software Development are often much smaller: Daily and continuous integration/testing paradigms shrink the full development cycle to a day, an hour, and possibly shorter. Continuous integration suggests continuous cycles of development and testing.

Delayed versus Immediate Feedback

For both optimum control and learning, "immediate feedback" is vital. Feedback from testing is much delayed in the waterfall model of development, somewhat delayed for end user during iterative cycles of development, and immediate for the developer in the continuous testing and continuous integration models of software development.

What is more, "immediate feedback" is also important to psychological well-being of the developer. Immediate feedback is a precondition for the "Flow" experience [26] that brings about a sense of being in control, performance at optimum capacity and productivity, and enjoyment that lasts through and beyond the duration of the activity.

Comprehension Of Evolving Software

Software comprehension is often considered to be part of reverse engineering and/or maintenance, and as such, not connected to testing. Exploratory testing is one testing paradigm that suggests that comprehension and testing can be performed in parallel.

Exploratory testing is an unscripted testing approach where the tester creates hypotheses and explores them with every tool that is in his/her arsenal. He/she may not only run the program, but may also generate code (often for automated testing), modify existing code (printouts or other), read code itself. The idea is that these tasks that are often segregated to different disconnected stages of software development actually work well together – in other words, there is synergy between software development stages.

2.2.3 Complexity: Software Size and Strong Components

Black Box, White Box, Information Hiding

In any given software, there are libraries, packages, classes and functions that are linked but happen to never be called from the software's executables. If the language allows dynamically generated calls (such as those created by reflection or dynamic linking), it may not be safe to strip the executables to remove any entities not statically needed in the software's executables.

Even in functions regularly called and needed, some branches may never execute due to patterns of input arguments used in this program. This means that the code base practically always contains code for more types of behavior than is exercised by the software during runtime. In fact, modularity, readability and reusability of packages, classes and functions mandate higher code complexity than minimally required (as could be satisfied with a monolithic cryptic program), and produce more generally usable components than is strictly necessary.

We consider complexity encountered during static analysis of the program "white-box complexity". As every allowed input for any function must be considered, this complexity is often more than the term "static" in "static analysis" suggests. In fact, due to unmanifested behaviors and unused functionality mentioned above, this white-box complexity is always greater than the black-box complexity that is encountered by running the software and interacting with it as end users do. For large code base, white-box complexity of the software can easily become unmanageable for a single developer.

Fred Brooks, the author of Mythical Man Month (MMM) [19] used to promote the idea of documenting the whole program (source code) and printing it and making it available to every developer as reference. 20th anniversary edition of MMM (1995) is not very different from original 1975 edition, except, in 20th anniversary edition, Fred Brooks extracted 240 implicit assertions of 1975 edition of MMM, all of which he believed still stood. One stark exception is information hiding. In a section titled "Parnas was right, and I was wrong about information hiding" [19, pp 271-273], Brooks conceded that due to sheer size of code base, his past practice of distributing all source code to every developer is not a feasible approach any more, and encapsulation and information hiding make more practical sense.

Complexity Analysis of Scenarios

It would take years to just read some of today's software code base once through, and there are various other issues of complexity that make complete comprehension of software as well as expected prior knowledge incomplete, for domain/vertical, environment (including OS and possibly hardware) and tools (including programming language and libraries). For further analysis of essential incompleteness of comprehension, see appendix B.2.

For large-scale software where it is not feasible for any one developer to read the whole source code even once, we can see that each task should be focused only on what is minimally needed so as not to require full comprehension of the code base:

- Testing, Verification: Unit tests and integration tests separate concerns. Modularity and use of modules, subsystems and components allow each part to be independently specified, verified and trusted. The standard structures of encapsulation and information hiding allow making "trust"/"verify" decisions per component.

- Analysis and Comprehension: Software comprehension should not be a pure top-down or a pure bottom-up process; comprehension of any function or object requires both dependents and dependencies to be known. Also, for any software that runs, dynamic behavior can greatly help comprehension task.

- Debugging: Has to focus on understanding a minimal slice of code sufficient to discover the bug. If there is no up-to-date feature-to-code mapping, a top-down traversal may be needed, especially in code not known to the developer, as proposed in Shapiro's 1983 thesis, "Algorithmic Program Debugging" [73].

- Specifications Discovery: Completeness of specifications should not be a goal. As this scenario is closely related to comprehension, specifications discovery cannot use a pure top-down or bottom-up traversal.

- Measuring Automated Verifier Adequacy: Specifications and documentation may be missing and not known by the developer. Best approach may be to use the source code itself to verify the verifier, if an approach similar to code coverage can be used.

2.2.4 Contract: The Requirement Specifications Document

In an ideal world, customers and end users would help shape, agree upon, and not demand any changes to their contract with the developer team. This contract is the requirements specification document.

Ideal Contract

If this contract (the requirement specifications document) exists and actually satisfies the end users, our scenarios become much simpler:

- Testing ≡ Verification (Validation is unnecessary, or it is just a formality)
- Analysis and Comprehension: User-level whole-system comprehension can be achieved by reading the specifications document. This still leaves out the developer's comprehension of implementation (including software architecture and design decisions).
- Debugging: Is made much simpler, and does not require trying to guess what behavior the end user requires of the program. Debugging still has to be performed, but contract is trusted, therefore comparing program behavior to contract is sufficient to verify program and find fault.
- Specifications Discovery: High-level requirements are already specified in the up-to-date requirements specification document (contract). Lower-level requirements specification discovery is helped to a degree, by the need to conform to contract wherever low-level implementation influences observable program behavior.
- Measuring Automated Verifier Adequacy: Automated verifier must minimally represent an automation of everything in the contract. This would be sufficient for end

users, but developers interested in reusability, maintainability, modifiability, and other qualities of software may also want to add verification of design and implementation, and verification of behavior of various parts of the software.

Automated Verifiers

If we could also create automated verifiers and a test suite that can check for all specified requirements, we would have:

- Testing ≡ Running the automated verifier on our test suite
- Analysis and Comprehension: All requirements can be learned by studying the contract (specifications document), the automated verifier, and the test suite.
- Debugging:
  - Locating fault origin: If individual parts of the software have verifiers, debugging can be greatly aided by turning on verification of parts during software execution, to discover faulty behavior before it manifests itself to the user as a defect.
  - Fixing the defect: Full verification automation would greatly aid in preventing introduction of secondary defects while eliminating the discovered defect.
- Specifications Discovery: Unnecessary at high level, as contract is correct. Necessary at low-level if automated verifiers at lower levelsof implementation are hard to follow and understand; otherwise, unnecessary or trivial.
- Measuring Automated Verifier Adequacy: This is still required, and would measure how accurately the automated verifier covers the requirements of the contract. If verifier must also cover behavior of parts, design and implementation as well, these must be checked separately against developer's documentation/requirements from the software.

2.3 Debugging: Strategies, Innovative Techniques, Efficiency

We stated earlier in section 2.1.3 that standard debugging paradigm has not changed much since 1961. In the meanwhile, program sizes and computer speeds have grown significantly. In the example we gave in that section, we saw a factor of 1.4 million more instructions per second.

In the following, we analyze debugging further, by:

- Discovering how standard debugging approaches of “step” and “run until” are not very reasonable in today’s much bigger programs and faster machines as compared to those in 1961.
- Looking at a simple example to see how simple print statements can help understand program behavior and help locate fault origin much more than standard debugging paradigm does.
- Doing a quick review of some innovative debugging techniques, and examining why they are not widely used.
- Envisioning the features of an ideal debugging tool.

2.3.1 Standard Debugging Strategies: Small vs. Large Steps

There are a few strategies that can be followed within the standard debugging paradigm:

1. Step: Debug one instruction at a time, sometimes checking program state
2. Skip: Execute a loop or function without interrupting, then pause execution for inspection

3. Run To/Until: Try to catch execution at a high-level line of instruction or event (some type of change in program state) In this case, the instruction may execute, or the event may occur frequently or infrequently. We may interrupt:

   - a. Somewhat regularly
   - b. Rarely
   - c. Never (wrong hypothesis about mechanism of failure)

4. A combination of the above strategies: Multiple breakpoints & registered tracked events.

We will see below that the first strategy ("step") cannot be sustained even for one second of program execution. The second strategy ("skip") may miss defect origin, and can only be employed for short periods; if done inside a repeating loop, it becomes the third strategy. In the third strategy ("run to/until"), cases a and b may miss defect origin, and case c would cause debugged program to run free to completion or crash, definitely missing the defect origin.

Unfortunately, stepwise nature of these strategies become more costly as computers run faster.

As we mentioned earlier, in 1961, DDT debugger ran on DEC (Digital Equipment Corporation) machines such as PDP-3, which could run 100,000 instructions per second [71] whereas 50 years later, in 2011, Intel i7 2600K runs at a speed of 128.3 billion Dhrystone instructions per second [41].

This means that on today's desktops, using the first strategy for one second of program execution would require pausing the program execution billions of times. If one second went through one billion steps, and we need to spend 5 seconds on average per step to check program state, this would take 5 billion seconds = about 700 work years of debugging [2].

[2]Assumption: 40 hour work week and 50 work weeks per year. Then, 5 billion seconds = 694.4 work years.

Scaling down, stepping through 1 milliseconds of program execution would take about 8.4 months. In one eight-hour work day, we can only step through 5.76 microseconds of program execution.

Clearly, we cannot sustain the first strategy ("step") of executing one instruction at a time for even a few microseconds of program execution today.

Debugging with such microscopic steps takes much time, and the significant feedback delays introduced cause:

- Fault localization to be an inefficient and slow process
- Learning/comprehension to suffer and be inefficient:
  - With longer delays between action and effect, it takes longer to learn from feedback, so learning/comprehension becomes less efficient.
  - Delays can cause errors in recall, making the developer unsure about any conclusions drawn.
- Lack of a sense of control, as quick feedback is essential to smooth control.

Catching program at more or less regular intervals may be more feasible, but it is likely that we will miss the execution of, and will not be able to infer the location of the defect in the source code. Interrupting a program 1,000 times per second of execution would have caught every 100th instruction in 1961, but would catch every one millionth instruction in the above scenario for 2011 (every 128 millionth "Dhrystone" instructions) In one million instructions, the code is likely to traverse a very large number of libraries, classes and functions, and it becomes very difficult to understand what happened in the interim, often making it impossible to guess the defect origin.

This means for both strategy 2 ("skip") and strategy 3, case a ("run until", with somewhat regular events), discovering the defect origin may be quite difficult. Note that the defect origin is a rare, singular event, and an event that occurs somewhat regularly cannot correspond directly to the defect origin.

When a combination of these strategies is employed, the advantages and disadvantages may be combined, but there may be too many pauses, and at each pause the developer has to recognize which type of breakpoint or event has caused the program to pause.

The best situation is strategy 3, case b ("run until" with rare events) if the event tracked corresponds to a correct guess (hypothesis) about mechanism of failure. A rare but unrelated event would not help at all.

Ideally, the event catches the internal state corruption at a point close to defect origin in the source code. If internal state corruption is discovered later in execution, this approach may have to be repeated with fresh restarts. With good hypotheses about mechanisms of failure, we may be able to go backwards in time, by restarting the debugger multiple times, to follow causal links backwards between executed source code and discovered markers of faulty system state.

Standard debugging paradigm does not support this case very well; it does not keep old program state, requires many fresh restarts, does not record hypotheses, and does not recall encountered events of interest in program execution.

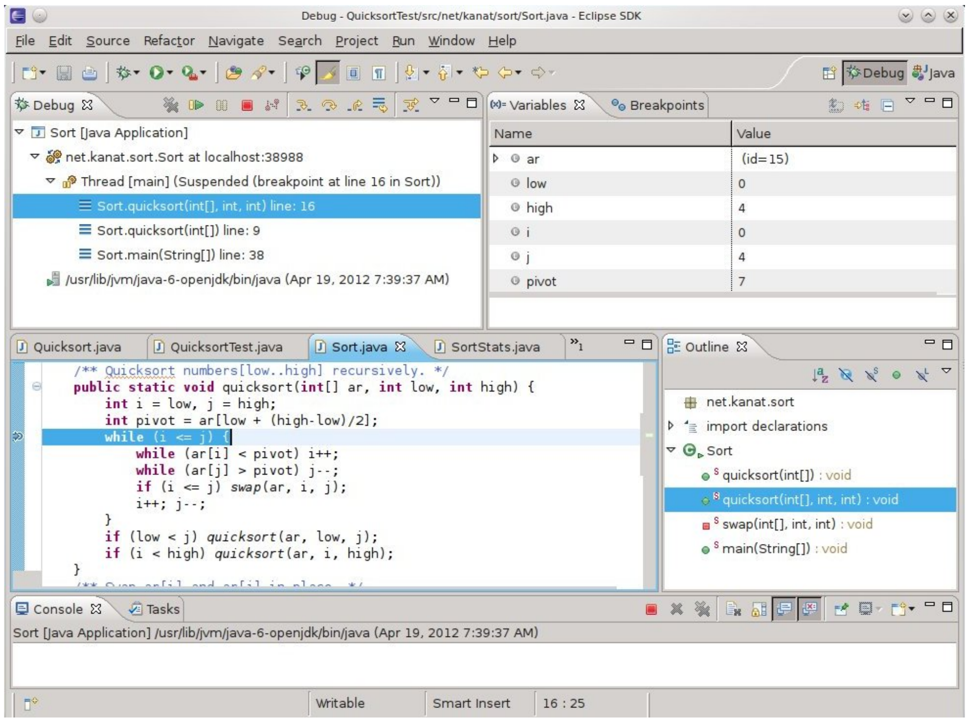
2.3.2 Standard Debugging For A Buggy Quicksort Function

Consider this quicksort implementation in Java:

```
public class Sort {
   /** Quicksort array ar in place. */
   public static void quicksort(int[] ar) {
      if (ar != null  &&  ar.length > 1)
         quicksort(ar, 0, ar.length - 1);
   }

   /** Quicksort numbers[low..high] recursively. */
   public static void quicksort(int[] ar, int low, int high) {
      int i = low, j = high;
      int pivot = ar[low + (high-low)/2];
      while (i <= j) {
         while (ar[i] < pivot) i++;
         while (ar[j] > pivot) j--;
         if (i <= j) swap(ar, i, j);
         i++; j--;
      }
      if (low < j) quicksort(ar, low, j);
      if (i < high) quicksort(ar, i, high);
   }
   /** Swap ar[i] and ar[j] in place. */
   private static void swap(int[] ar, int i, int j) {
      int temp = ar[i];
      ar[i] = ar[j];
      ar[j] = temp;
   }
}
```

This quicksort implementation has a small bug; instead of sorting the array {5, 1, 7, 4, 2} properly, it returns {1, 2, 5, 4, 7}. It does not always fail, and actually works well for most arrays of this size, and many larger arrays as well. For example, this quicksort properly sorts [14, 3, 19, 12, 2, 7, 10]. For arrays without duplicate values, this algorithm correctly sorts 100% of arrays with three elements, 60% of arrays with five elements, and 15.4% of arrays with 10 elements.

Fig.2.1 is a screenshot of a popular Java IDE, Eclipse, being used to debug this program, at the start of the first loop, for values {5, 1, 7, 4, 2}. On this screen, after expanding "local variables" view to see low, high, i, j, and pivot, there is not enough space to see the array



```
Sort [Java Application]
  net.kanat.sort.Sort at localhost:38988
    Thread [main] (Suspended (breakpoint at line 16 in Sort))
      Sort.quicksort(int[], int, int) line: 16
      Sort.quicksort(int[]) line: 9
      Sort.main(String[]) line: 38
  /usr/lib/jvm/java-6-openjdk/bin/java (Apr 19, 2012 7:39:37 AM)
```

Name	Value
ar	(id=15)
low	0
high	4
i	0
j	4
pivot	7

```java
    /** Quicksort numbers[low..high] recursively. */
    public static void quicksort(int[] ar, int low, int high) {
        int i = low, j = high;
        int pivot = ar[low + (high-low)/2];
        while (i <= j) {
            while (ar[i] < pivot) i++;
            while (ar[j] > pivot) j--;
            if (i <= j) swap(ar, i, j);
            i++; j--;
        }
        if (low < j) quicksort(ar, low, j);
        if (i < high) quicksort(ar, i, high);
    }
```

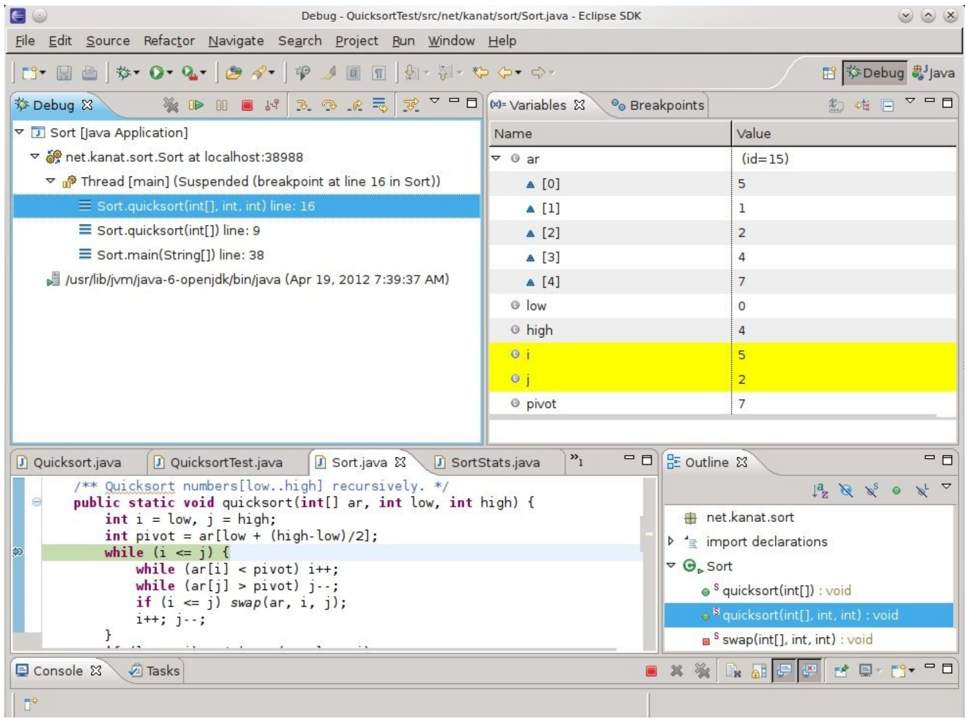
```
net.kanat.sort
import declarations
Sort
  quicksort(int[]) : void
  quicksort(int[], int, int) : void
  swap(int[], int, int) : void
  main(String[]) : void
```

```
Sort [Java Application] /usr/lib/jvm/java-6-openjdk/bin/java (Apr 19, 2012 7:39:37 AM)
```

Figure 2.1: Debugger running on buggy quicksort algorithm, with ar = {5, 1, 7, 4, 2}, before executing while loop the first time.

values as well (that are displayed vertically).

Debugging is exploratory. The place we know there is a problem is when we observe failure, which is beyond the defect origin. As shown in the previous section, there is no easy choice of strategy with a standard debugger. We can start by stepping or skipping the while loop. If we cannot see the array, we cannot understand what is going on. On a standard debugger, we can never see array values from two separate points of execution time together, so we have to remember older values. Eclipse highlights changed array elements with yellow background, which is helpful. But we still do not know what values these array elements had before they changed, and we cannot rewind execution once change takes place. By the time new value is highlighted with yellow background, the previous value is lost, and the only way to discover the old value may be to restart the debugger and go through the same steps we went until now.


Debug - QuicksortTest/src/net/kanat/sort/Sort.java - Eclipse SDK

```
Sort [Java Application]
  net.kanat.sort.Sort at localhost:38988
    Thread [main] (Suspended (breakpoint at line 16 in Sort))
      Sort.quicksort(int[], int, int) line: 16
      Sort.quicksort(int[]) line: 9
      Sort.main(String[]) line: 38
  /usr/lib/jvm/java-6-openjdk/bin/java (Apr 19, 2012 7:39:37 AM)
```

Name	Value
ar	(id=15)
[0]	5
[1]	1
[2]	2
[3]	4
[4]	7
low	0
high	4
i	5
j	2
pivot	7

```
    /** Quicksort numbers[low..high] recursively. */
    public static void quicksort(int[] ar, int low, int high) {
        int i = low, j = high;
        int pivot = ar[low + (high-low)/2];
        while (i <= j) {
            while (ar[i] < pivot) i++;
            while (ar[j] > pivot) j--;
            if (i <= j) swap(ar, i, j);
            i++; j--;
        }
```

```
net.kanat.sort
import declarations
Sort
  quicksort(int[]) : void
  quicksort(int[], int, int) : void
  swap(int[], int, int) : void
```

Figure 2.2: Debugger running on buggy quicksort algorithm, with ar = {5, 1, 7, 4, 2}, after defective instruction has run (internal state is now corrupted).

Fig.2.2 is a screenshot at the time of defect origin. Ordinarily, the developer does not know where defect origin is, and would not know to pause program execution at this exact point, out of hundreds or possibly thousands of points we could pause execution at during the execution of this program with these inputs.

To see the array, we had to have fewer source code lines visible, and we collapsed console output view completely as well. Last instruction changed i and j, so they are highlighted in yellow. Their values are inconsistent with our quicksort implementation expectations, so program internal state is now corrupt. But neither the program nor the debugger tell us anything about this program's expectations (internal state consistency requirements). As program state history is not revealed, it is also hard to know what the responsibility of this function is during this call, and what falls outside this function's responsibility.

Even when we pause the program at the exact point it fails, right after the defective code executes, it may still be very difficult to see that the program has failed.

During stepwise debugging, this failure will likely not be noticed, and program execution would be continued. If program state is found to be corrupt at a later time, program has to be started again under the debugger to discover fault origin.

2.3.3 Print Statements

Many developers still prefer simple print statements because unlike stepwise debugging, print statements:

- collectively display information about multiple points in execution, revealing a high-level picture.
- can be followed backwards in time to go from revealed failure back to first case of unusual program internal state.

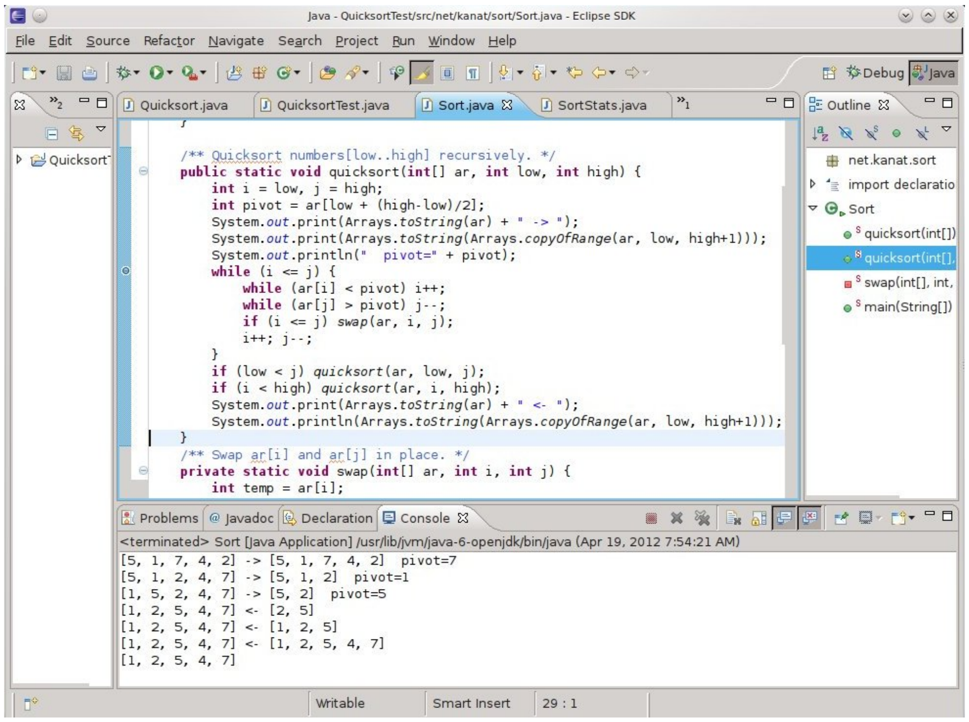
Even though print statements often reveal a small portion of the program state, compared to standard debuggers, they can be much easier to use and much more efficient in discovering mechanisms of failure and defect origin.

Let us now see in the previous quicksort example how print statements compare to using the debugger.

Fig.2.3 shows program with print statements run to completion. As we did not start the debugger, Eclipse uses standard Java editing view, which gives us more space for code even when we view all print statements. In this case, we have used costly simple standard Java functions rather than defining any helper functions; we create a new array just to print the part of the array, each time. Compared to the overhead of debugging step-by-step, this cost is insignificant on today's computers.

There is no need to guess when defect may occur, no need to pause execution or step through. Printed values show program status on each quicksort entry and exit:

```
[5, 1, 7, 4, 2] -> [5, 1, 7, 4, 2]  pivot=7
[5, 1, 2, 4, 7] -> [5, 1, 2]  pivot=1
```



```
    }

    /** Quicksort numbers[low..high] recursively. */
    public static void quicksort(int[] ar, int low, int high) {
        int i = low, j = high;
        int pivot = ar[low + (high-low)/2];
        System.out.print(Arrays.toString(ar) + " -> ");
        System.out.print(Arrays.toString(Arrays.copyOfRange(ar, low, high+1)));
        System.out.println("  pivot=" + pivot);
        while (i <= j) {
            while (ar[i] < pivot) i++;
            while (ar[j] > pivot) j--;
            if (i <= j) swap(ar, i, j);
            i++; j--;
        }
        if (low < j) quicksort(ar, low, j);
        if (i < high) quicksort(ar, i, high);
        System.out.print(Arrays.toString(ar) + " <- ");
        System.out.println(Arrays.toString(Arrays.copyOfRange(ar, low, high+1)));
    }
    /** Swap ar[i] and ar[j] in place. */
    private static void swap(int[] ar, int i, int j) {
        int temp = ar[i];
```

```
<terminated> Sort [Java Application] /usr/lib/jvm/java-6-openjdk/bin/java (Apr 19, 2012 7:54:21 AM)
[5, 1, 7, 4, 2] -> [5, 1, 7, 4, 2]  pivot=7
[5, 1, 2, 4, 7] -> [5, 1, 2]  pivot=1
[1, 5, 2, 4, 7] -> [5, 2]  pivot=5
[1, 2, 5, 4, 7] <- [2, 5]
[1, 2, 5, 4, 7] <- [1, 2, 5]
[1, 2, 5, 4, 7] <- [1, 2, 5, 4, 7]
[1, 2, 5, 4, 7]
```

Figure 2.3: Print statements to see the big picture. We can traverse backwards from observed failure, to understand the mechanism of failure, and to find the defect origin.

```
[1, 5, 2, 4, 7] -> [5, 2]  pivot=5

[1, 2, 5, 4, 7] <- [2, 5]

[1, 2, 5, 4, 7] <- [1, 2, 5]

[1, 2, 5, 4, 7] <- [1, 2, 5, 4, 7]

[1, 2, 5, 4, 7]
```

We use "->" for entry to quicksort, and "<-" for exit from quicksort, which should accompany a sorted subarray. We print both the full array (easier to compare and follow) and the part of the array that quicksort is focused on. We also print the pivot value – in our implementation, this is always the element in the middle of the input subarray.

The final array {1, 2, 5, 4, 7} is not sorted because of the fourth element, 4. As we see the program state history, we can see that this element never moved from its original location. Ordinarily, we would go backwards to see who is responsible from having 4 in the wrong place. In our simple example, this brings us all the way up to first printed line, which corresponds to entry to first call of quicksort. From first to second line, we can see that pivot operation is correctly performed for 7, but 4, which is less than 7, should have been included on the left subarray to be sorted. Instead of the first four elements, only the first three elements, {5, 1, 2}, are sorted.

Our left partition ends at index j (inclusive), which traverses array from the right until the test condition $i <= j$ fails. It appears that j is decremented more than it should be; it travels too far to the left. Maybe the $j--$ statement (and by symmetry, probably also the $i++$ statement) should be removed? That would change behavior of quicksort for any input, which sounds like a somewhat radical change in implementation.

But we also know that our sort algorithm works on some other arrays. Actually, looking at our printouts, we can see that both {5, 1, 2} and {5, 2} are sorted properly by our quicksort. So the error must be conditional; j travels too far left under some conditions, but travels the right amount under other conditions. We may suspect the only if statement in the while loop, $if(i <= j)$, and $j--$ is outside of that if statement. The bug is that incrementing i and decrementing j at the end of while loop should be inside the $if(i <= j)$ branch, not outside:

```
while (i <= j) {
   while (ar[i] < pivot) i++;
   while (ar[j] > pivot) j--;
   if (i <= j) {
     swap(ar, i, j);
     i++; j--;      // bug fixed
   }
   // i++; j--;   <--  bug location
}
```

Once the code is “instrumented” with useful print statements, running the program is very quick, and the output can be quite informative. Using print statements is a low-tech approach to the question of fault localization, and yet it has a number of advantages over standard debugging methods.

2.3.4 Some Innovative Debugging Techniques

Various innovative techniques have been proposed that can improve the efficiency of discovering the defect origin. Reverse debugging techniques address the issue of locating defect origin starting from the later point in program execution where the bug manifests itself as an observed failure:

- **Reversible/Bidirectional Debugging**: Allows reverse-execution by regularly saving the whole program state; disk-space intensive [17].
- **Omniscient Debugging**: Reverse execution and an event search query language [58].
- **Whyline**: Generates "Why ...?" and "Why not ...?" questions, and can answer them, going backwards in execution [48].

Such reverse execution approaches use a method called "checkpointing" which captures full program execution state at frequent intervals. Checkpointing has very high memory and disk-space costs, and may slow down a program by a factor of 10 or more. For example, Whyline for Java [48] reports overhead factors of 4.1 - 14.3 compared to normal program execution, and this factor was observed when a very small program was loaded on a computer with very large memory so that there would be no delays due disk swap space usage.

Except for "Whyline", the interface remains low-level debugging, which can make it difficult to see patterns of program behavior. In bidirectional debugging, program history can be changed after going backwards in time. This allows testing different execution branches, but can also be confusing after a few forward and backward executions with history change. Other approaches (Omniscient Debugging and Whyline) do post-mortem analysis of program state for one program execution, so program cannot be interacted with according to knowledge gained by these debugging techniques – for example, a hypothesis about why program has failed cannot be tested by entering different inputs to running program (this could be done in stepwise debugging).

There's no adoption in the industry for these approaches mainly due to various reasons of inefficiency and inadequacy:

- There's very high memory, disk space and CPU time overhead
- Most research tools are only demonstrated on very small target programs (mainly due to memory overhead)
- Significant change of timing can change program execution due to real-time timeouts within code
- Reversible approaches cannot revert environment side-effects (such as deleting a file, writing to any stream, etc), so only self-contained programs can be reverse-executed.

Another innovative approach is Delta Debugging:

- **Delta Debugging**: Finds minimal input difference or internal variable value difference that causes failure.

The original delta debugging focuses on the inputs to the program [82] [85]. Later, this approach was extended to include analysis of internal variable values in the program [83] [23], which allowed delta debugging to analyze internal causes of state change and program failure.

Delta Debugging reruns program many times with different inputs/values. This can take much time, but does not need much disk space. It requires one good and one bad input, ways to modify the inputs, and a verifier. Delta Debugging keeps trimming the difference between the inputs, and ends when it finds the smallest possible change in input that makes a difference between correct and failed execution. Even though this is called "debugging", the focus is on data, and combing through code is done manually by developer after Delta Debugging returns minimal input difference.

In [83] and [23], the same idea is instead applied to variable values that make up the internal program state (that is not usually visible to end user or even the caller of a function). Causal chain delta debugging [23] attempts to automate discovering causal links, to find fault origin from observed failue. Even though conceived as an approach for full automation, it currently is an assistive technology that causes a reduction in the number of lines of code to examine to find fault origin.

In tests, this method pinpointed the fault origin in 4.65% of tests. 30% of the time, the developer needed to examine 10% of the code, whereas 55% of the time the developer needed to examine 25% of the code. Even though this reduction is helpful, this can still be a lot of code to examine, and human analysis of causal links and high-level structure of the program can often also quickly eliminate large portions of code from need for further analysis.

Andreas Zeller summarizes these approaches in his 2005 book titled "Why Does My Program Fail?" [84]. This book was mentioned as "soon to appear", in 2002 [85] by the book title "Automated Debugging," and in 2005 [23] by the book title "Why does my program fail? A guide to automated debugging." It seems that Zeller had to drop the claim to have automated and therefore solved the problem of debugging.

Some other innovative techniques that help indirectly are:

- **Static Slicing**: Finds lines of code potentially responsible from a variable's value [11].
- **Dynamic Slicing**: Finds lines of code responsible from a variable's value, during one execution [3] [2].
- **Query-Based Debugging**: Uses queries to catch when program internal state becomes corrupt [56] [55] [39] [40] .

Slicing helps improve focus while examining code, but without reverse execution, it is still possible to lose state and be unable to discover defect origin going only forward in time. Query-based debugging requires planning ahead and compiling program with queries. Queries help much only when they are custom-written for the bug in question. Going backwards through causal links to discover prior mechanisms of failure may require program to be recompiled and rerun with different different queries.

2.3.5 Envisioning An Ideal Debugging Tool

Standard debugging paradigm runs efficiently, but is inefficient in use as it focuses on microscopic steps compared with program execution time. Reverse execution techniques look very promising when applied to very small target programs, but they are very inefficient in both memory and speed as they need to store very large amounts of program state data regularly during program execution.

Not all researched approaches are practial to apply to large-scale software. Much can be learned by attempting to convert a research technique to scale up to industrial-scale software with a reasonable level of efficiency and interactivity. Boshernitsan et al. discovered some practical lessons while implementing Agitator [18], an experimental industrial implementation of some testing research ideas. Agitator team discovered by experience the requirements for an automated testing tool intended for efficient and relatively easy operation by an average developer, and we believe the same requirements also apply to an automated and practical debugging tool. Such a tool should:

- partially automate first step of debugging, discovering defect origin
- be efficient
- have wider focus:
    - help see dynamic context of execution & program state
    - lead to exploratory rather than linear behavior

Beyond these general requirements, an ideal debugging tool would also:

- use intuitive and/or familiar forms of interaction
- apply recent innovative methods of debugging
  - may need to use efficient heuristics that approximate results rather than use original inefficient algorithms
  - may need to process software at certain points and times instead of complete processing
- integrate with familiar tools (for example Eclipse or NetBeans for Java development)

Chapter 3

Techniques and Technologies

This chapter introduces the various techniques and technologies used in VERDICTS and SMT. In the next few sections, after explaining how the different techniques work together in VERDICTS and in SMT, we will look at each technique and technology used in VERDICTS and SMT.

3.1 VERDICTS: How DBC, AspectJ, Beanshell and Statistical Views Work Together

VERDICTS is an approach to testing and verifying software. It is designed to support dynamic exploration/investigation of a complex software system. VERDICTS research implementation accomplishes this by inserting contracts (class and method requirements/verifiers) using interception, interpreting contracts written in Java, while collecting statistics in a flexible way to provide multiple views of software dynamic behavior.

VERDICTS research implementation uses:

- Contracts: Design by Contract (DBC) using Java statements (3.3)
- Interception: Aspect-Oriented Programming (AOP) with AspectJ (3.4)

- Interpreter: Beanshell, a live Java interpreter (3.5)

- Statistics & Visualizations: Box plots, correlation matrix, etc (3.6)

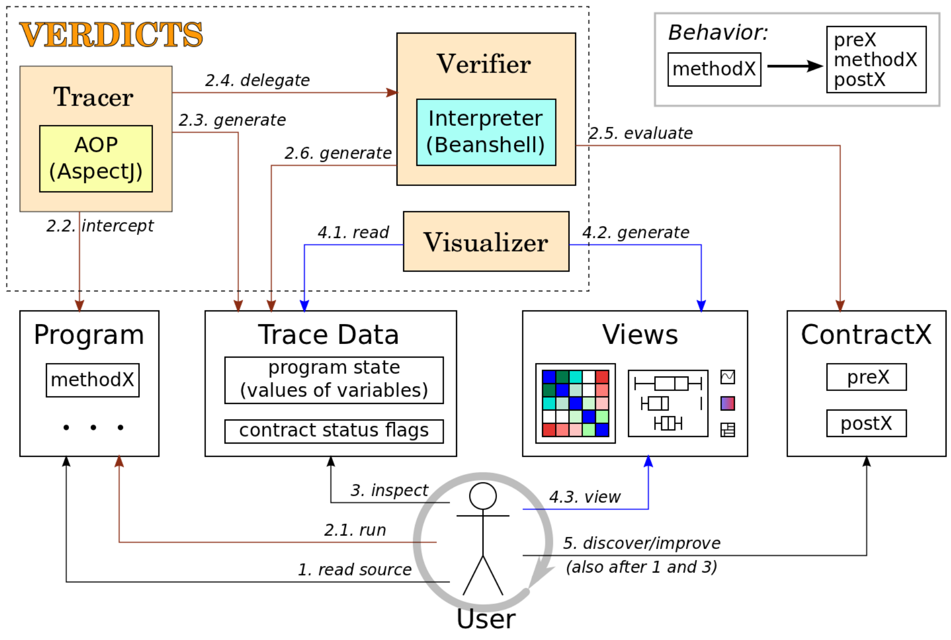


Figure 3.1: VERDICTS process is an analyse-hypothesize-test cycle. Techniques and technologies used are the DBC Contracts, AOP (Aspect-Oriented Programming), Interpreter, Statistics and Visualization (as Views above)

In VERDICTS, the unit of test is a method. To understand a method and discover its contract (DBC contract), the user can read the source code of the method (if available) as seen in step 1 of fig.3.1 for methodX, to discover contractX.

Reading source code is useful, but for nontrivial methods that depend on many other methods which may also have to be understood, reading source code of all methods involved is not an efficient way to discover what one method attempts to deliver. Dynamic behavior analysis provides an efficient complementary approach to understanding a method.

As seen in fig.3.1, steps 2.1 - 2.6, VERDICTS adds the ability to run the program and observe its dynamic behavior via direct inspection of trace data and contract status flags (step 3) as well as by viewing statistics and visualizations (steps 4.1 - 4.3). VERDICTS Tracer uses AOP (AspectJ) to intercept the method entry/exit events and store trace data.

After the user hypothesizes an initial version of contractX for methodX, VERDICTS Tracer calls the Verifier, which in turn uses Java Interpreter (Beanshell) to evaluate the contract and store more trace data (user-defined variables) and contract status flags. All steps of this process can be repeated in a cycle to improve the initial guess for method contract.

AspectJ allows dynamic load-time weaving, which means that any library or a jar file with many classes can be loaded and set up for event interception just before execution.

VERDICTS allows developers to create method verifiers by writing method contracts in the original programming language used in implementing the components (Java), because this is the most familiar language for a developer to use. This code for verification is then executed live by using Beanshell Java interpreter. More powerful language features of Beanshell and OCL (Object Constraint Language) need not be known to be able to use VERDICTS, but can be used by experienced developers.

As DBC can be used as an automated testing oracle, contracts can also be used to test parts of a system within the framework of the integrated system (tested “in situ”). However, within a system, a component may not be tested fully, if its inputs do not span the full breadth of allowable inputs for the component. For more thorough and focused testing of a method in isolation, the interactively discovered contracts of that method can be exported and combined with a richer set of test inputs. This richer set of test inputs can combine inputs gathered during component runtime with manually generated and pattern-based automatically generated inputs to the method. Inputs combined with contract can then serve as an automated unit tester, and could replace use of test suites and manual tests.

3.2 SMT: How ASM, Reflection and Class Reloading Work Together for Semantic Mutations

SMT (Semantic Mutation Testing) is our technique to evaluate contract quality with respect to a program component. In SMT, code is first analyzed for data flows and dependencies

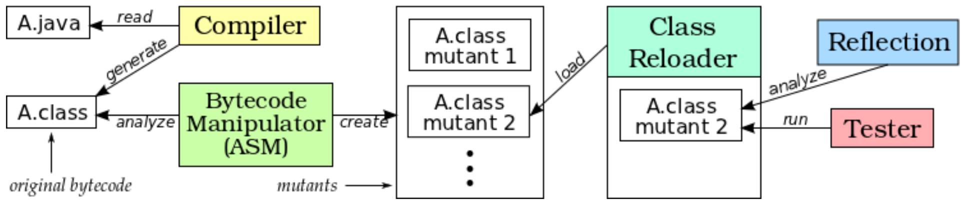


Figure 3.2: SMT techniques and technologies: Mutation Testing using Bytecode Analyzer/Manipulator (ASM), Class Reloader, and Reflection (Java).

between method arguments, fields, local variables and constants, then modified (mutated) to create small changes in program state while avoiding some common types of crashes.

SMT approach is based on traditional (syntactic) Mutation Testing (MT, also called Mutation Analysis). Our implementation in Java uses:

- Error injection: Ideas from traditional MT (3.7)
- Analysis: Java reflection (3.8)
- Modification (Java bytecode manipulation): ASM library (3.9)
- Dynamic reloading: Java class reloading (3.10)

As seen in fig.3.2, bytecode analysis and manipulation library (ASM) allows us to generate mutants (mutated versions of original bytecode) for a Java class that can be loaded with a class reloader to use standard Java reflection and our tester on each mutant. Important points to note:

- ASM library allows us to make arbitrary modifications to Java classes (bytecode)
- Java class reloading gives us dynamic replacement of executing code.

Using Java class reloading and ASM, we can test various modifications to a Java class with respect to some functional or nonfunctional requirements or metrics. Modification of bytecode and reloading can happen completely in memory rather than on the filesystem, and mutants can be generated and stored one at a time, making this process very efficient.

3.3 Design by Contract (DBC)

3.3.1 Human-Language Specifications vs. DBC

Human-language specifications are pervasive, but also often incomplete, ambiguous, and inconsistent. From a six-line requirements specification by Naur that at first sight appears to be clear, unambiguous and complete, even after two research publications where specifications were claimed to have been corrected and made complete ([34] and [35], by Goodenough and Gerhart), Meyer [60] still shows errors of incompleteness, ambiguity, inconsistency and incorrectness.

In object-oriented programming, Meyer suggests using "contracts" that formally declare obligations between methods. Class and method contracts consist of:

- Method Precondition: What the method requires from any caller before starting execution.
- Method Postcondition: What the method ensures upon termination.
- Class (Object) Invariant: The object internal consistency conditions that must be preserved (and can also be expected) by all nonstatic public methods.

Meyer calls pervasive use of contracts "Design by Contract" (DBC). In essence, the method contract is postcondition minus precondition: The contract for a method is:

- IF the caller satisfies preconditions before call,
- THEN the method will satisfy postconditions upon termination,

Nonstatic public methods must also preserve the object invariant. This means, for those methods, invariants are added to both the precondition and the postcondition.

For example, consider this method with description that can be used as human-language specification:

```
/** Returns truncated squareroot of n for n >= 0.
 *  For example, sqrt(15) == 3, sqrt(16) == 4.
 */
int sqrt(int n) { ... }
```

There's an assumed but unspecified requirement: returned value is the nonnegative squareroot. A good contract in Java language, using $r to hold return value from the method, is [1]:

```
pre: n >= 0
post: $r * $r <= n
post: ($r + 1) * ($r + 1) > n
```

With DBC, cross-method communication formats and expectations can be documented unambiguously, even before implementing the methods. Both during integration testing, and as components are added or modified, DBC speeds up discovery of issues and breakage, as these quickly appear as contract failures.

If a precondition fails, the responsible party is the caller. If the precondition does not fail but the postcondition fails, the method itself is responsible and the method implementation is faulty.

Separating these tests from the method implementation allows such method contracts to be turned on/off individually or en masse as needed. Meyer suggests that the contracts are:

- All turned on (always executed) for component testing
- Selectively turned on for integration testing (usually, preconditions are turned on, postconditions and invariants are turned off)
- All turned off for deployment (after having had much testing)

[1]Note that even though $\$r \geq 0$ is not explicitly stated, it is still required by the combination of the two postconditions stated here; for integer $\$r$, $(\$r + 1)^2 > \$r^2$ implies $\$r \geq 0$.

As preconditions usually contain much simpler tests, this approach allows fast integration testing to discover inter-component miscommunication. Ability to turn all contracts off easily allows fast execution after deployment. Meyer is against writing input data validity checks in method code. This practice can add too much validity checking, and can slow down a system. Putting this instead in precondition for a method allows quick global switching of validity checks.

A few languages support DBC natively, most notably, Eiffel (by Meyer himself) and D. Most popular programming languages have third-party tools that provide DBC support [30]. In Java alone, [30] lists 18 third party tools that provide DBC support. Appendix D compares three relatively popular DBC specification languages, Eiffel, OCL (Object Constraint Language, part of UML), and JML (Java Modeling Language). In this document, we use OCL keywords "pre" and "post" for brevity, combined with Java language declarations (instead of OCL) for familiarity.

3.3.2 Our Proposal: Selective Retrofitting of Contracts

The idea of using specifications and specifically DBC contracts, in the creation of automated test oracles has been studied before [24]. Our approach differs in proposing selective retrofitting of discovered and/or evolved contracts.

Standard DBC paradigm suggests using contracts from the start and to use them in all components. But this is not necessarily an efficient method. In two controlled experiments, Mueller et al. discovered that DBC improved cross-developer code reuse, reliability, and maintenance efficiency, but the initial development phase took longer [62]. But delaying the initial development has the disadvantage of delaying user feedback that can be used to improve the system quality and usability.

Instead of starting with full-coverage and pervasive contracts even before implementation, we suggest an approach that allows selective retrofitting of contracts onto a system developed (or otherwise acquired) with no contracts. Similar to use of profiling to improve efficiency of selected functions, our approach can be used to focus first on the classes most central to program's operation. This way, important gains in maintainability may be achieved without significant start-up costs.

3.3.3 Programmatic DBC using Java Statements

In research implementation of VERDICTS, we use DBC to specify requirements. Adding the ability to use other forms of executable (and therefore automatically verifiable) specification would not require major changes to the design of VERDICTS. If there is an existing converter from a specification language to Java, the Java code can be interpreted live by VERDICTS, and we would only need to add a small amount of code for integration with VERDICTS.

Eiffel, OCL and JML are three popular declarative and partially-declarative languages for DBC mentioned earlier in section 3.3.1 (see examples given in appendix D).

The problem with these languages is that most programmers are not familiar with them, and there is often a steep learning curve before being able to declare anything beyond the simplest contracts. VERDICTS instead starts with programmatic DBC with Java statements that are executed to verify method preconditions and postconditions.

Programmatic syntax has the advantage of being familiar to developers; no new formal specification language has to be learned. Gary Leavens who worked on Larch, a formal language with algebraic declarative syntax, later headed the more programmatically specified JML (Java Modeling Language) [53] [54] mainly because of developers' familiarity and comfort with the programming languages as opposed to algebraic specifications.

Programmatic syntax also has the obvious advantage of being directly executable, and as such, has unambigious semantics to any developer when compared to a higher level specification language.

Programmatic syntax may, in certain cases, have the disadvantage of being more verbose. For example, standard for loops are often more verbose than "forall" and "exists" specifications, especially when nested. Verbosity may hinder clarity. Allowing a set of common macros and common functions can quickly improve the readability of programmatic DBC, and make it very close in power to declarative DBC.

In the future, we would like to experiment with usability of OCL (Object Constraint Language), JML (Java Modeling Language), and other DBC specification languages that allow more concise specification of expected behavior. Often, these specifications are convertible to executable Java code, so these can be implemented as a layer on top of current functionality.

3.4 Aspect-Oriented Programming (AOP) With AspectJ

3.4.1 What is AOP?

Aspect-Oriented Programming (AOP) is a programming paradigm that increases modularity by separation of "cross-cutting concerns". Cross-cutting concerns often go across layers of abstraction, so they cannot in general be modularly implemented at any layer. AOP allows quickly attaching, enabling and disabling an action to common events and situations that appear in a large number of locations in the source code. A simple example is logging.

Gregor Kiczales and colleagues at Xerox PARC developed the concept of AOP [47], as well as the AOP framework AspectJ [7] for Java, which remains the best known AOP framework today. Today, most popular languages (Java, C, C++, .Net languages, Ruby) have either native implementations or external libraries for AOP. Even some non-programming languages have external libraries to implement AOP: MAKAO for make, Motorola WEAVER for UML 2.0, and AspectXML for XML [6].

In order to separate cross-cutting concerns, we need a way to declare points of interception in program, and what to do at these points of interception. This depends on:

- join points: Places in code to intercept execution; events of interest
- pointcuts: Expressions that define sets of join points, often by using regular-expressions

and wildcards on names, locations (package/module) and signatures (function call signature, field declaration signature, etc)

- advice (name used in AspectJ): Statements to execute at join points

Most commonly used join point type is for method entry, exit, or both. With “around” advice for a method, for example, the whole method can be intercepted and its behavior changed: Method body may be skipped under some conditions, any other code can be executed, and return values can be changed. Some other events of common interest are object creation and destruction, access or modification of a field, and exceptions (when a Throwable gets thrown in Java).

3.4.2 AspectJ and Dynamic AOP

Our implementation uses AspectJ to intercept method calls.

In AspectJ, pointcuts and advice are combined into aspects that resemble Java classes. Each aspect may affect bytecode of every class, except for security-protected standard Java library code. This is accomplished by “weaving”, which inserts calls to advice code (as source code or bytecode) into the program at every matching join point that the advice applies.

Static weaving happens at compile-time. The more powerful “dynamic load-time weaving” (LTW) can directly manipulate bytecodes of any class loaded into memory. Through LTW, AspectJ aspects can intercept and affect the behavior of any class in a third-party library for which we only have class files (often compiled into JAR files), and no source code. Even calls between methods in the library can be intercepted.

AspectJ does not allow the aspects or pointcuts to be modified during runtime. This feature, called “dynamic AOP”, would require either:

1. using tie-in code that allows switching from one instrumented version of the class to another version, or,

2. unloading and reloading the class.

The first approach requires that client code be written from the start for such instrumentation. This is used by some dynamic AOP frameworks through the use of Java dynamic proxies which are reflection-based proxies that multiplex all method calls of a class through a single method. These frameworks require that client code use the framework's interfaces and create objects that may be instrumented using only the framework's factories. This approach cannot instrument third party code.

The second approach is the most generally applicable approach. Unfortunately, no Java AOP framework today uses class reloading. As the first approach limits clients and the second approach is not available today, no Java AOP framework today supports dynamic AOP on any given client class. There is one way to emulate dynamic AOP on any given class:

1. User specifies at the start the superset of methods that may be intercepted.
2. During runtime, user can switch interception of any of these methods on or off.
3. During runtime, user can also switch which/what advice applies to each method.
4. A common advice is used for the superset of methods. This tie-in code does the necessary demultiplexing (like a switchboard):
    - It discovers quickly if interception for this method is currently on.
    - If on, it delegates to proper advice specific for this method.

For VERDICTS, this approach has proven to be efficient, without imposing a significant performance penalty or degradation in responsiveness for the client code [13].

There is an alternative to AOP frameworks for a commercial tool that is interested in dynamic interception of method entry/exit events: The Java Virtual Machine Tool Interface (JVMTI).

JVMTI is a low-level programming interface that combines Java's historically separate profiling and debugging interfaces. JVMTI allows very detailed registry of events of interest with a JVM, and its use would allow efficient dynamic instrumentation of Java code. With JVMTI, tracing and data collection must be done with C or C++ code. Presentation/view layer could still be written in Java, possibly with communication to the C/C++ back-end (data layer) through a socket connection.

3.4.3 Testing Unit: Method

In OOP, methods are usually the smallest unit of interest for testing. For example, JUnit considers each tested method xyz(...) as a black box, with only the inputs and outputs visible to test method. In JUnit naming convention, Abc.xyz(...) method is usually tested with AbcTest.testXyz() method. In JUnit, as the tested method is considered a black-box, its calls to any helper methods are not intercepted. If those helper methods must also be tested, they are tested separately, elsewhere, in separate test methods.

In our work, we are also interested in methods as units of test. For DBC precondition and postcondition evaluation, we need to store information and verify internal state of program before and after execution of methods of interest. With AOP, this means that we are mainly interested in method call entry and exit join points. One difference from JUnit is that AOP does not consider intercepted method as black-box: From an intercepted method, calls made to other methods of interest will also be intercepted, and helper method behavior will also be verified. As AOP intercepts methods of interest independently, multiple method entry events can be intercepted before the first method exits. AOP can also intercept private methods. All this matches very well with what is expected from a DBC framework.

3.5 Live Java Interpreter, Beanshell

3.5.1 Compiled vs. Live Interpreted Java Code

Java started its life as an interpreted language. JVM is a stack-based machine, and Java programs are distributed as Java bytecode, which contains instructions that run on this stack machine.

For a given architecture, it makes more sense to convert these instructions to system-native instructions. To keep Java programs portable, Java programs are still distributed as architecture-agnostic Java bytecode, but bytecode is compiled just-in-time (JIT compilation) before it is executed the first time.

To make VERDICTS dynamically usable, we need to be able to insert Java statements (or statements in another language for specifications and operations) while a program is running. In order to be able to do this, we use interpreted Java, using Beanshell interpreter. For each method tested with preconditions and postconditions, we use a separate Beanshell interpreter object, to efficiently set and segregate method contexts.

Alternatively, we could use Java class source code or bytecode generation and dynamic loading/reloading. This type of Java watch expressions and evaluation expressions are used in probably all IDEs and debuggers, and a similar non-Java implementation (possibly reuse of open source Eclipse or Netbeans implementation) may prove more efficient in a commercial version of VERDICTS.

3.5.2 Beanshell

Beanshell is a small, embeddable Java source interpreter that also allows common scripting language features beyond standard Java syntax. It can be used as an interpreted Java shell console. In VERDICTS, we use Beanshell to interpret programmatic DBC method contracts made up of precondition and postcondition declarations written in Java.

Unfortunately, Beanshell is not actively maintained, and so it does not allow any recently added Java language capabilities. The latest version of beanshell, dated 5/23/2005, can be downloaded at the Beanshell download page. Thanks to Java remaining backwards compatible, Beanshell still works well with the latest version of Java.

3.5.3 Beanshell Features, Examples

Beanshell can evaluate standard java declarations, expressions and statements, including code blocks, loops, method calls, and object creation. Beanshell extends Java language with language features of scripting languages, most notably, loosely typed variables, and ability to define global functions. These features make Beanshell more convenient as they reduce the amount of typing required for an action.

In standard Java, we may use this code to display a button in a frame using Swing classes JButton and JFrame:

```
String label = "Click Here";
JButton button = new JButton(label);
JFrame frame = new JFrame("Actionless Button");
frame.getContentPane().add(button, "Center");
frame.pack();
frame.setVisible(true);
```

In Beanshell, we can skip the declarations; types will be discovered from values:

```
label = "Click Here";
button = new JButton(label);
frame = new JFrame("Actionless Button");
frame.getContentPane().add(button, "Center");
frame.pack();
frame.setVisible(true);
```

Beanshell also has a few global helper functions as Beanshell commands, which are really predefined Beanshell scripts. For example, `frame()` displays a GUI component in a frame (JInternalFrame, JFrame or a Frame, as available and applicable), so the code above can be further shortened (if we can ignore the frame title) to:

```
  label = "Click Here";
  button = new JButton(label);
  frame = frame(button);
```

Some other bsh commands are `pwd`, `cd`, `cp`, `mv`, `rm`, `cat`, `editor`, `exec`, `eval`, `super`, `print`, `run`, `save`, `load`, `setFont`, `show`, `javap`, `source`, `exit`. Beanshell also allows users to extend this command set by defining other Beanshell scripts and placing these user-defined Beanshell scripts in the class path.

Even though Beanshell has a number of other scripting features, the main reason we have chosen to use Beanshell is that it allows live evaluation in a language that is familiar to the end users. For the most part, the end users never need to know that Beanshell has any features beyond Java. All that matters is that the familiar syntax of Java language allows users to write any Java code and have it interpreted and evaluated.

Behind the scenes, VERDICTS is free to use extended features of Beanshell interpreter, to make life easier for the end user. This allows VERDICTS to support OCL, which can be used by more experienced/advanced users.

3.6 Statistics & Visualizations

We gather program behavior data by saving values held in variables in the program as well as user-defined "observables" that are very similar to "watched expressions" used in debuggers. As we use methods as unit of testing, variables and data gathered also have one method as their declaration context. Over time, a population of data [2] is gathered for each variable of each method.

To visually depict data and statistics, we use some standard statistical descriptive methods:

[2]Note that as Java uses pass-by-reference, object state is not preserved unless a clone is explicitly created by the user – this would be executed each time the method is called. As this can get very expensive, making copies of some fields will often be preferable.

- X-Y plot, to see relationship between any two variables
- Correlation matrix, to notice relationships between all pairs of variables
- Box plot, to see degree of dispersion (spread) and skewness for individual variable

Other than these, VERDICTS also uses a large number of types of OO and GUI visualizations; we will describe these in detail later as part of VERDICTS.

Any reader should already be familiar with X-Y plots. Readers familiar with statistics are probably also familiar with the concepts of correlation matrix and box plot as well. These descriptive statistics approaches are easy to follow even for novice readers. For example, exact definition of boxes and whiskers need not be known to intuit that it represents dispersion/spread of data. The next two subsections define and give examples of box plots and correlation matrices.

3.6.1 Box Plot (Box and Whiskers Diagram)

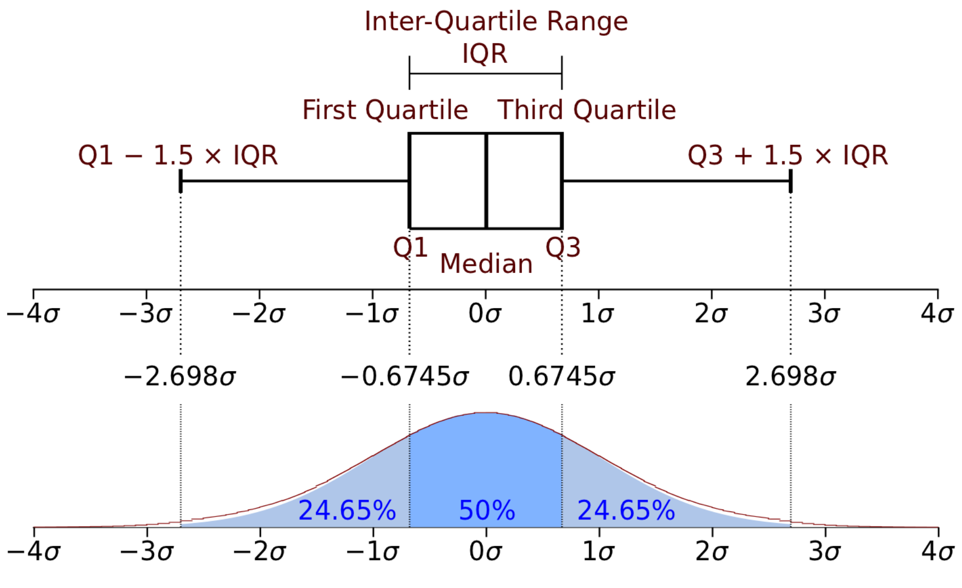


Figure 3.3: Expected box plot for (large) data set with normal distribution, without the outliers. Number of outliers is expected to be proportional to population size, which is not specified here.

A box plot (also called box and whiskers diagram) graphically depicts the dispersion of numerical data including the five-number summary (min, max, and the three quartiles) and outliers (unexpectedly high or low values). Fig.3.3 shows expected box plot for large data set with normal distribution without the outliers.

Two "whiskers" that are depicted as terminated line segments that extend from the boxes show the spread of data. In a finite population, the whiskers are often farther in from the cutoff values shown in fig.3.3; they instead show the lowest value (in the data) $\geq Q1 - 1.5 \times IQR$ and the highest value $\leq Q3 + 1.5 \times IQR$.

Any value in the data that does not fit within the whiskers is considered an outlier, and is separately depicted in the box plot (not shown in fig.3.3), usually with an open circle or a cross mark. For large data set where a wider spread is more likely by chance, wider whiskers are sometimes used to only consider and highlight unexpectedly far values as outliers.

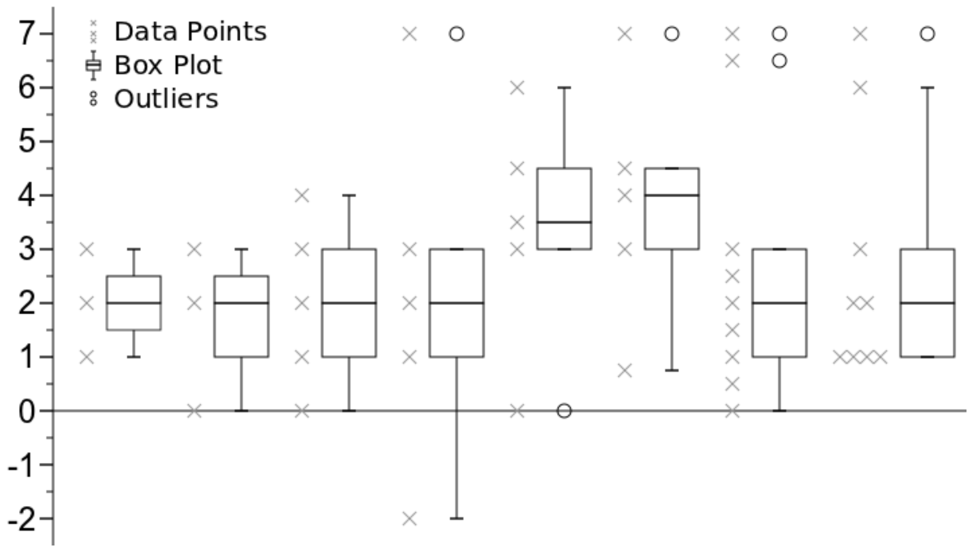


Figure 3.4: Some data shown with X marks and corresponding box plots (using empty circles for outliers).

Fig.3.4 shows box plots for some data sets. Data sets are shown with cross marks, and outliers for box plots are shown with empty circles in this figure. Note that whiskers are sometimes missing (often, terminated on the box), because there is no data within 1.5 IQR distance from the box. The same data and corresponding box plot parameters are also shown numerically in table 3.1.

Table 3.1: Data and corresponding box plots' boxes (Q1, Q2, Q3), whiskers (W1, W2) and outliers.

Data	...	...	...	Data	W1	Q1	Q2	Q3	W2	IQR	outliers
1	2	3			1	1.5	2	2.5	3	1	
0	2	3			0	1	2	2.5	3	2	
0	1	2	3	4	0	1	2	3	4	2	
-2	1	2	3	7	-2	1	2	3	3	2	7
0	3	3.5	4.5	6	3	3	3.5	4.5	6	1.5	0
1	3	4	4.5	7	1	3	4	4.5	4.5	1.5	7
0, 0.5	1, 1.5	2, 2.5	3	6.5, 7	0	1	2	3	3	2	6.5, 7
1, 1	1, 1	2, 2	3	6, 7	1	1	2	3	6	2	7

For any method of interest, after one tracing session of data collection, VERDICTS uses a compact horizontal representation of box plot for each variable/observable of interest.

Instead of the customary vertical alignment, VERDICTS uses horizontal alignment as seen earlier for normal distribution in fig.3.3. This allows VERDICTS to fit a box plot to same height as one line of text that describes the variable. VERDICTS uses a color-coded distribution density visualization as background for the box plot. Outliers are shown with vertical lines to allow for a more compact representation.

3.6.2 Correlation Matrix

We show correlations between pairs of variables/observables (probe variables, arguments, return values, etc) of interest using a color-coded correlation matrix visualization.

A correlation matrix of random variables X_1, X_2, ... X_n is the symmetric table of correlation coefficients between pairs of these variables. We use Pearson correlation coefficient, which is the standard deviation-normalized covariance of two variables X and Y, where covariance is the expected value of joint deviation of X and Y from their mean, as seen in equation 3.1:

$$\rho_{X,Y} = corr(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{3.1}$$

In essence, correlation coefficient normalizes the covariance by individual variances, and therefore ignores the spread of data. Correlation coefficient is always within -1.0 and 1.0.

As correlation coefficient is a single real-valued number with a limited range, it cannot reveal the type of nonlinearity of relationship between two variables, but merely that two variables are not completely linear in their relationship.

A correlation matrix is simply a table of correlation coefficients, between all pairs of variables. As this table is always symmetric and the main diagonal is filled with the correlation coefficient 1.0 (corr(X, X) is always 1.0), it is sufficient to only show one half of the matrix.

Table 3.2: Four data sets with same basic statistics (modified from Anscombe's quartet; y0 is the regression line).

x	4	5	6	7	8	9	10	11	12	13	14
y0	5.0	5.5	6.0	6.5	7.0	7.5	8.0	8.5	9.0	9.5	10.0
y1	5.39	5.73	6.08	6.42	6.77	7.11	7.46	7.81	8.15	12.74	8.84
y2	3.10	4.74	6.13	7.26	8.14	8.77	9.14	9.26	9.13	8.74	8.10
y3	6.90	6.26	5.87	5.74	5.86	6.23	6.86	7.74	8.87	10.26	11.90

Consider the four data sets with same set of X values in table 3.2 and fig.3.5. We generated these sets from "Anscombe's quartet", the four data sets created by statistician Francis Anscombe in 1973 [5] to have the same statistics but different behavior. Our four data sets, $y_0 \ldots y_3$, each have:

- $mean(y_i) = \mu_{y_i} = 7.50$
- $stddev(y_i) = \sigma_{y_i} = 2.03$
- the regression line $y = 0.5x + 3.0$

Even though they have same/similar statistics, these data sets behave differently:

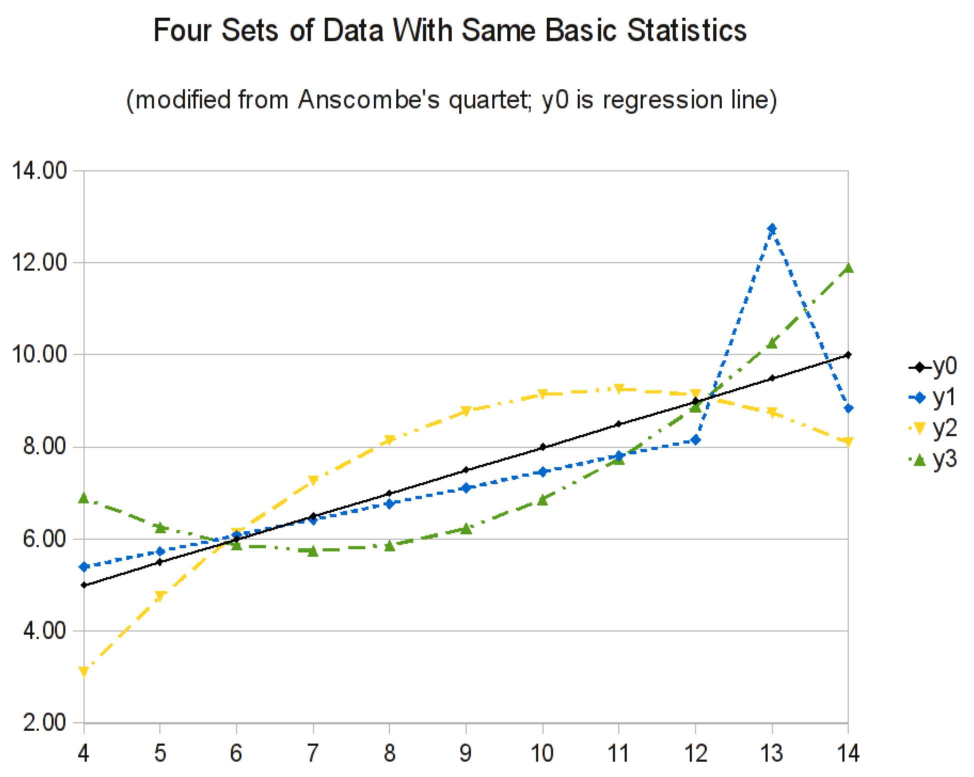
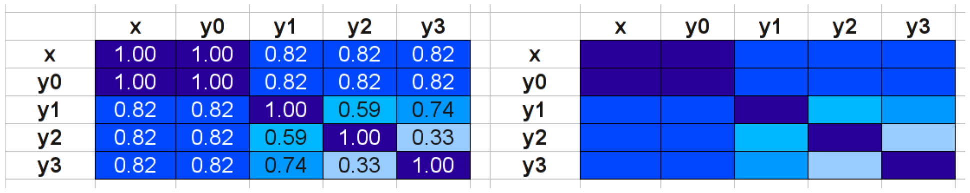


Figure 3.5: Four data sets with same basic statistics (modified from Anscombe's quartet).

- $y_0 = 0.5x + 3.0$ has linear relationship with x. This is the regression line for $y_0 \ldots\ y_3$.
- y_1 is linear except for one high-valued outlier
- y_2 appears like a hill plus incline, and is Anscombe's nonlinear curve example.
- y_3 appears like a valley plus incline, and has y values of y_2 reflected around y_0; $y_3(x) = 2y_0(x) - y_2(x)$

y_0 has perfect linear relationship with x, and therefore, has correlation coefficient $corr(x, y_0) = 1.0$. The others (y_1, y_2, y_3) have $corr(x, y_i) = 0.82$.

Correlation matrix can be represented visually using color and/or saturation to show correlation, on a two-dimensional grid. In visualization, converting numbers to colors and therefore converting a numerical table to a grid of colors in this way is known as using a “heat map”. For a correlation matrix, each grid box has uniform color, the grid is symmetric around the main diagonal, and the main diagonal shows strongest color, for correlation coefficient = 1.0.



	x	y0	y1	y2	y3
x	1.00	1.00	0.82	0.82	0.82
y0	1.00	1.00	0.82	0.82	0.82
y1	0.82	0.82	1.00	0.59	0.74
y2	0.82	0.82	0.59	1.00	0.33
y3	0.82	0.82	0.74	0.33	1.00

Figure 3.6: Correlation matrix shown with numbers and colors, and with only colors.

Fig.3.6 shows the correlation matrix for the four data sets shown above (x with $y_0 ... y_3$), using background color (still showing correlation coefficients numerically) and with uniform fill colors, using lightness of blue to represent lack of correlation. This example does not have negative correlation. A different color, for example red or green, could be used to depict negative correlation.

Note that y_3, which is a reflection of y_2 around regression line y_0 has the least correlation with y_2. Reflection adds some negative correlation (when y_2 goes down, y_3 goes up, etc), but shared regression line with y_2 still keeps some positive correlation (when y_2 is min, y_3 is not max, etc).

In VERDICTS, beyond color, the correlation matrix is also be overlaid with representations of constancy, along x, y or both dimensions. If x is constant, x vs. y plot will be a vertical line. If both x and y are constant, the plot is simply a single point. If $x \equiv y$, the plot is a simple 45-degree line, and the correlation coefficient is 1.0. This case happens along the diagonal of the correlation matrix as each $X_i \equiv X_i$.

As a final note, we would like to remind that correlation does not imply causation. Correlation is symmetric, so the direction of causation could never be deduced. There might also be interim variables that form a causal chain or a common cause for both x and y. Still, correlation is one of the best indicators of deterministic dependence between variables.

3.7 Mutation Testing (Mutation Analysis)

Our SMT approach to measure contract adequacy depends on mutation testing, Java reflection, bytecode manipulation and class reloading. This and the next three sections will go through these techniques and technologies. Please refer back to section 3.2 to see how they work together in SMT.

We consider the traditional mutation testing (MT) as per [65] to be syntactic mutation testing. More appropriately called "mutation analysis" (of test suites), this method does not test software, but rather provides a test criterion to evaluate test suite adequacy (quality), similar to various code coverage criteria.

For an original program P which passes test cases T_1, T_2, ..., mutation testing uses a predefined set of mutation operators (such as '/' $\rightarrow$ '+') to create "mutants" M_1, M_2, ... which are all possible single-mutation versions of P. Mutants are compiled and checked against the test suite. If mutant M_i does not pass the test suite (if any test case T_j fails), we consider mutant M_i killed. Otherwise mutant M_i remains "live".

A mutant is semantically equivalent to P if it always behaves the same way as P for any input. No test case (that passes P) could kill an equivalent mutant. Mutation adequacy score is the number of killed mutants divided by the number of non-equivalent mutants. If this score is 1.0, all non-equivalent mutants are killed by our test suite, and the test suite is called "mutation adequate".

As we mentioned earlier, researchers report a number of problems with MT (quoted from section 5.5):

- MT has high computational complexity, mostly due to compilation of each mutant program.
- Semantic equivalence of mutants to original program is a tedious, manual task [36]
- Many mutants are so different in behavior that they may fail almost all tests. [44] had about 41% of all mutants fail all tests, and called these "dumb mutants".

- “Competent programmers write programs that have few defects” does not mean competently written programs have small syntactic differences. Jia and Harman [44] call this misconception “Syntactic-Semantic Size Myth”.

For a more detailed critique of MT, see section 5.4.

Because of these issues, we propose Semantic Mutation Testing, which focuses and limits semantic difference from original (unmutated) program.

3.8 Java Reflection

Java reflection allows Java programs to introspect themselves. Classes can examine themselves or each other, to discover and access fields, methods, attributes, annotations,

The first step in using reflection is getting a Class object that represents the class we are interested in. Then we can look at its fields, methods, constructors, annotations, and inner classes:

```
import java.lang.reflect.*;
...
String className = "java.lang.Integer";
Class cls = Class.forName(className);   // or cls = Integer.class
for(Field       fld : cls.getDeclaredFields())  ...
for(Method      mtd : cls.getDeclaredMethods())  ...
for(Constructor ctor: cls.getDeclaredConstructors())  ...
for(Annotation  ann : cls.getAnnotations())  ...
for(Class   innerCls: cls.getClasses())  ...
```

Reflection can cause various Throwables/Exceptions to be thrown due to illegal access or lack of element with desired name or structure (such as method signature), so the code above and below should ideally be put in a try-catch block.

There are four ways to access class substructures (fields, methods, constructors, and annotations). Two methods take no arguments and return the whole arrays of substructures:

- getDeclared...s(): Gets all ... (fields/methods/constructors/annotations) declared directly in this class, including not only public but also protected, package-visible and private substructure elements.
- get...s(): Gets all publicly visible ... whether they are defined in this class or inherited from superclasses or implemented interfaces.

Two other methods are very similar but not pluralized in name, and require arguments (such as name and signature) to help identify a single substructure of the class:

- getDeclared...(...): Gets one uniquely identified substructure (possibly not public) that is defined in this class (either not inherited, or overwritten).
- get...(...): Gets one uniquely identified, publicly visible, possibly inherited substructure.

To get all non-public substructures of some type, we need to use getDeclared...s() methods for all superclasses (direct and indirect ancestors) and possibly all interfaces/superinterfaces if the class substructure we are interested in may be declared in an interface.

Using reflection, a constructor can be invoked to create an instance of the class under reflection. The following code snippet creates an object using a constructor that uses a single int argument:

```
Class cls = ...
Constructor ctor = cls.getConstructor(new Class[] { int.class });
Object o = ctor.newInstance(new Object[] { 3 });  // auto-boxed
```

We can check to see if an object is an instance of the class cls. If it is, we can call (invoke) its methods, read and modify its fields. The following code snippet increments by 1 values of all integer-valued fields declared in this class, and then calls all no-argument methods declared in this class, including all private methods, but excluding inherited methods that are not overwritten.

```
Class cls = ...
Object o = ...
if (cls.isInstance(o)) {
  for(Field fld: cls.getDeclaredFields())
    if (fld.getType() == int.class)
      fld.setInt(o, fld.getInt(o) + 1);
  for(Method mtd: cls.getDeclaredMethods())
    if (mtd.getParameterTypes().length == 0)
      mtd.invoke(o, new Object[] { });
```

By default, reflection respects visibility modifiers, but this restriction can be relaxed for programs executing locally. If the restriction is not relaxed, any private method and field access in the code snippet above would cause a SecurityException to be thrown due to illegal visibility-ignoring access to object internals.

For more information on reflection, please see The Reflection API Trail [43] in Java Tutorials.

Compared to ASM bytecode manipulation, reflection is a lightweight approach that can be used to get some information about loaded classes, and get information and modify objects. Reflection cannot be used to modify the class itself, whereas ASM library can change the class arbitrarily; it can change instructions that make up a method, and it can add or remove methods, fields, and internal classes.

3.9 ASM Library: Java Bytecode Manipulation and Data Flow Analysis

3.9.1 ASM Library for Java Bytecode Manipulation

We use ASM library in SMT to discover and logically prevent mutations that may cause program to crash such as ArrayIndexOutOfBoundsException. A small problem in single-mutation mutants, such crashes can debilitate multiple-mutation mutants generated by SMT. To this end, we use ASM to perform data flow analysis to find risky and safe variables to mutate.

ASM library is a Java bytecode engineering library that is embedded in a number of popular Java development tools. It can be used to parse and modify Java bytecode using two APIs:

1. Core API: An event-based sequential access parser API (similar in operation to SAX parser for XML)
2. Tree API: A memory-resident parse tree API (similar to DOM for XML)

ASM also supplies classes based on Tree API that help perform:

- data flow analysis (forward or backward analysis)
- control flow analysis

Similar to SAX parser for XML, event-based parser provided by Core API is lightweight, efficient, but requires forward processing. Any methods, declarations and instructions encountered later in the bytecode cannot influence processing of earlier methods, declarations and instructions, unless parsing is done in multiple passes.

Tree API produces a memory-resident parse tree in the memory, which allows for much more complicated analysis and processing of the bytecode, but also takes more CPU time and memory.

The next section explains data flow analysis by ASM as used in SMT. See appendix F for more details on ASM Core API, Tree API, and data flow analysis.

3.9.2 ASM Library for Data Flow Analysis

ASM data flow analysis is built atop Tree API, and allows our own "Value" objects to hold symbolic representations of all possible values that may be held by a variable, a field or a position in the stack.

As a forward data analysis example, consider this static slicing question: "Which lines may affect this variable's value at this line?" We can use Value objects to record and accumulate line numbers of all value sources that affect any value in any variable, field or stack position. The Value object for our variable at our target line will be the union of all possible line numbers that can affect this variable's value at this line.

Backward data analysis requires an extra processing step. Consider this question: "Is this local variable's value ever directly or indirectly used as array index?" This is indeed the question that SMT must answer to avoid array index out-of-bound crashes. In this case, we need to do backward analysis from array index use(s) back to local variable declaration, possibly indirectly through assignments and operations that pass and modify the value originally held in the local variable. To achieve this, we use Value objects in a forward pass provided by ASM to discover sources of influence, then process these Value objects in a second pass to find all uses that relate to declarations.

3.10 Dynamic Replacement of Executing Code With Java Class Reloading

Another interesting feature of the Java language is that the code that executes always belongs to a class, and classes can be loaded into memory using standard system class loader or our own class loaders.

This allows definition of a class loader that can unload and reload different implementations (compiled into different bytecodes) for a class any number of times during a program's execution. The just-in-time (JIT) compiler allows any compiler optimizations such as inlined linking between classes to be broken and recreated with a new definition of a class.

This is a very powerful feature. JUnit, for example, uses class reloading to reload and retest modified classes without restarting the JUnit GUI. This ability also allows Java to be used more easily for mutation testing. Many mutated variants ("mutants") of a class can be generated either by recompilation of changed source code, or by direct manipulation of bytecode to insert mutations. Mutants do not have to rename the class; they can be loaded one at a time, tested, and unloaded, as seen in fig.3.2.

Chapter 4

VERDICTS: Visual Exploratory Requirements Discovery and Injection for Comprehension and Testing of Software

4.1 Overview of the VERDICTS Approach, Process Cycle

4.1.1 VERDICTS Process Cycle and Core Components

VERDICTS is our research tool designed to test and demonstrate the value of using exploratory contracts in software analysis and testing. It combines software dynamic analysis and automated testing into a single dynamic rapid-feedback cycle that allows verification of hypotheses about software requirements and behavior.

VERDICTS proposes using contracts for both analysis and automated testing, and is designed to support contract discovery, creation, evolution, revisioning and evaluation. These

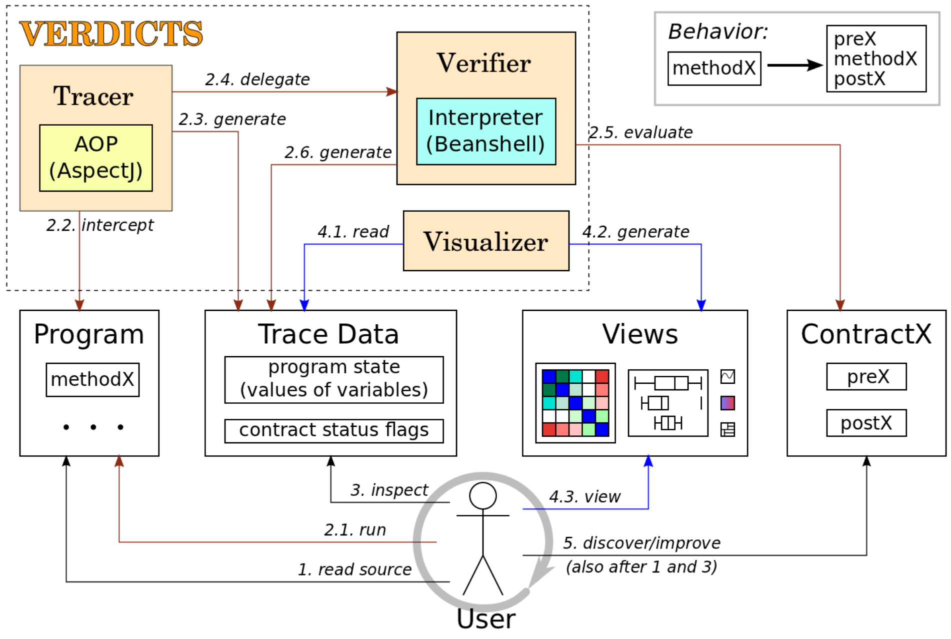


Figure 4.1: VERDICTS core components (Tracer, Verifier, and Visualizer) and process; figure copied from chapter 3. VERDICTS process is an analyze-hypothesize-test cycle.

contracts can be dynamically discovered and evolved in cycles of observation of behavior and modification of contracts as seen in fig.4.1 (figure copied from chapter 3).

Steps 1 - 5 form VERDICTS analyze-hypothesize-test cycle. If software source code is available, a developer can perform step 1, "read source code" without using any developer tools. If a debugger is available, we can also perform limited dynamic analysis.[1]

Steps 2 - 5 of VERDICTS, seen in fig.4.1 will be explained in more detail in sections 2 - 5 of this chapter. Before there are any contracts, steps 2-5 allow a user of VERDICTS to discover software dynamic behavior, and to record hypotheses of requirements and behavior of methods of interest. Once the user creates a contract, the same steps allow the user to test contracts and hypotheses against methods of interest. When a method's contract fails a method call, the fault may be in the method's implementation or in the contract. Further analysis and testing can reveal fault origin when it is ambiguous from the context. More

[1]As mentioned before in sec 2.1.3, a debugger gives a very limited view into the dynamic behavior of a program. As debugger does not record or recall program state over the execution, it cannot be readily used to discern patterns of behavior by comparing behavior at different times of execution. Debugger does not make unusual behavior stand out.

often, either the software or the user is trusted (for analysis and testing tasks, respectively), and blame is assigned by default to the other party; user's hypotheses and contracts during analysis, and software itself during testing.

Fig.4.1 also shows that VERDICTS achieves these goals through its three core components:

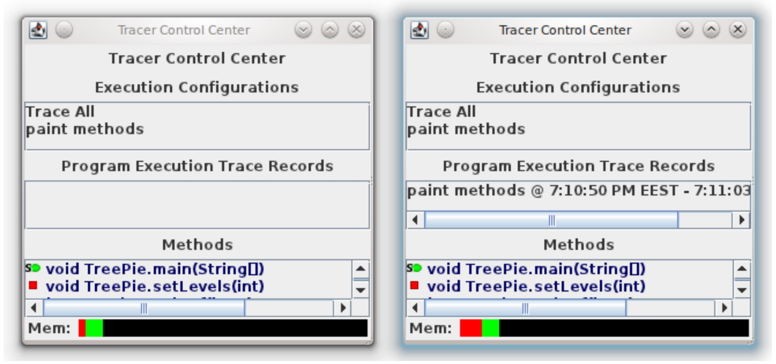
- **Tracer**: Interception, program state data collection
- **Verifier**: Contract evaluation
- **Visualizer**: Presentation of program state data and contract pass/fail status using various visualizations ("views")

4.1.2 Views and Visualizations

One simplification we have used in this figure is considering all views as visualizations. VERDICTS actually uses three types of views to inspect the program state data collected during one tracing session:

- *Trace views*: These views show method calls over time, possibly visually marking caller-callee relationships.
- *Standard debugger views* (Individual views): These views show values of variables at one method call entry and exit. The variables recorded are the arguments, the "$this" object, the return value, and any user-defined variables for the method.
- *Aggregate views*: These views are visualizations of data gathered across multiple method calls, to see patterns of behavior.

The first two types of views are covered in section 3, as we consider these to be part of step 3, inspecting tracing session, program state, and contract status flags. Only the third type of view is covered under section 4, as visualizations, for step 4.2, viewing the visualizations.



Tracer Control Center

Tracer Control Center

Execution Configurations

Trace All
paint methods

Program Execution Trace Records

Methods

void TreePie.main(String[])
void TreePie.setLevels(int)

Mem:

Tracer Control Center

Tracer Control Center

Execution Configurations

Trace All
paint methods

Program Execution Trace Records

paint methods @ 7:10:50 PM EEST - 7:11:03

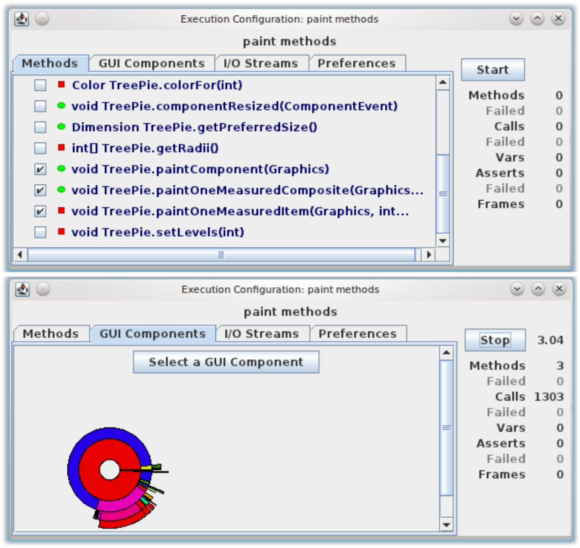
Methods

void TreePie.main(String[])
void TreePie.setLevels(int)

Mem:

Figure 4.2: VERDICTS Control Center (main window) shows tracing configurations, tracing sessions, methods and memory usage (used/free/max).

VERDICTS eliminates the need to recompile or restart the target system between versions of contracts. This approach supports quick feedback from hypotheses to tests, improving the comprehension and testing task efficiency. One execution of VERDICTS allows for multiple tracing sessions to be run. Fig.4.2 shows VERDICTS Control Center before and after the first tracing session is run.



Execution Configuration: paint methods

paint methods

Methods | GUI Components | I/O Streams | Preferences

- [ ] Color TreePie.colorFor(int)
- [ ] void TreePie.componentResized(ComponentEvent)
- [ ] Dimension TreePie.getPreferredSize()
- [ ] int[] TreePie.getRadii()
- [x] void TreePie.paintComponent(Graphics)
- [x] void TreePie.paintOneMeasuredComposite(Graphics...
- [x] void TreePie.paintOneMeasuredItem(Graphics, int...
- [ ] void TreePie.setLevels(int)

Start

Methods	0
Failed	0
Calls	0
Failed	0
Vars	0
Asserts	0
Failed	0
Frames	0

Execution Configuration: paint methods

paint methods

Methods | GUI Components | I/O Streams | Preferences

Select a GUI Component

Stop 3.04

Methods	3
Failed	0
Calls	1303
Failed	0
Vars	0
Asserts	0
Failed	0
Frames	0

Figure 4.3: VERDICTS Execution Configuration window controls list of methods to be traced, GUI components to be recorded, and execution trace to be started/stopped (on the right).

List of methods traced and GUI components followed can be changed between tracing sessions by defining execution configurations. Fig.4.2 lists two execution configurations named

“Trace All” and “paint methods”.

Fig.4.3 shows the Execution Configuration window for “paint methods” configuration. This was created by cloning the original “Trace All” configuration of VERDICTS, modifying it to only trace paintXYZ methods, making it track a GUI component and renaming the configuration to “paint methods”. In this case, the target program is a Treepie/Sunburst type visualization of disk space usage of a directory, and the GUI component tracked is the full GUI of the target program. Original “Trace All” configuration and this configuration are listed in VERDICTS Control Center, and can be reached from there.

The second Execution Configuration window screenshot in fig.4.3 is taken after selecting the GUI component and clicking the “Start” button to start the tracing session. 1303 calls for three methods have been traced at the time of this screenshot. The tracing session has not been stopped yet.

4.1.3 Step 1: Read Source Code

The final word on what software does, and how it works is always the source code. If source code is available, it is the first step in VERDICTS cycle as well. But as this is not the only step for VERDICTS, test/analysis task can move on to use of VERDICTS as soon as the developer is curious to examine or test one method’s dynamic behavior, possibly along certain inputs and outputs, possibly with a theory about how the method functions.

4.2 Step 2: Run Target Program under VERDICTS Tracer & Verifier

Step 2 of fig.4.1 is “run”. The user starts VERDICTS, and runs executables of the target software under VERDICTS by creating and starting an execution configuration as shown above. The user can then interact normally with the target software while VERDICTS accumulates runtime behavior data and compares behavior against contracts for methods of interest.

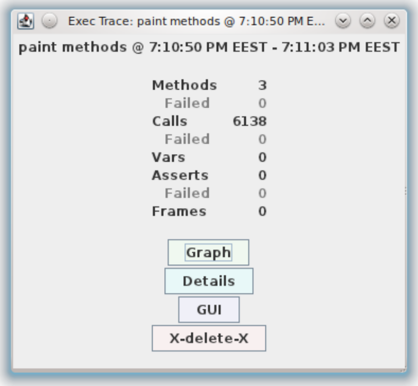
Tracer in VERDICTS goes beyond a traditional tracer that is only focused on recording function execution times to discover performance bottlenecks. It is also used to collect program state information by recording values of inter-method communication and user-defined contract variables at method entry and exit points. As Java is pass-by-reference, and does not allow every object to be cloned (does not allow creating copies of objects), VERDICTS tracer only records references. As we will see later, where cloning/copying is allowed by the language, user-defined variables can be used to create copies of objects. Excessive use of copies for low level frequently used methods that share large objects would obviously cause significant memory and speed penalties. VERDICTS allows users to decide how much data should be recorded in a tracing session. For further discussion on this topic, please see section 4.6 below.

As seen in fig.4.1, step 2 ("run") consists of six smaller steps:

- 2.1. User runs and interacts with the target Program
- 2.2. Tracer intercepts the target Program
- 2.3. Tracer generates Trace Data
- 2.4. Tracer calls the Verifier (delegates verification tasks)
- 2.5. Verifier evaluates the contract for each method called (contract contains user-defined variables and assertions)
- 2.6. Verifier generates more Trace Data (values of user-defined variables and contract status flags)

Before contracts are added, only steps 2.1 - 2.3 (Tracer) run, and trace data about execution time and inter-method communication variables (method arguments, "$this" and the return value) are recorded. After contracts are added, steps 2.4 - 2.6 (Verifier) also run. This adds values of user-defined variables and status of assertions in user-defined contracts.

A tracing session is started and stopped from its Execution Configuration window. Once stopped, the session will appear in the VERDICTS Control Center (main window) as seen previously in fig.4.2 with “paint methods” configuration. Clicking on the tracing session opens a session summary and view control window such as the one seen in fig.4.4.



paint methods @ 7:10:50 PM EEST - 7:11:03 PM EEST

```
Methods      3
  Failed     0
Calls     6138
  Failed     0
Vars         0
Asserts      0
  Failed     0
Frames       0
```

Figure 4.4: VERDICTS Trace Session View gives a summary and can open other views (windows, visualizations).

In this figure:

- The trace session summary shows the number of user-defined variables and assertions in method contracts, and how many assertions, calls and methods have failed. Any failed assertion fails the call, and any failed call fails the method.

- **Graph** button opens a window that shows “Methods Graph” which displays methods traced showing the call direction and frequency, using a force-based layout algorithm (see section 4).

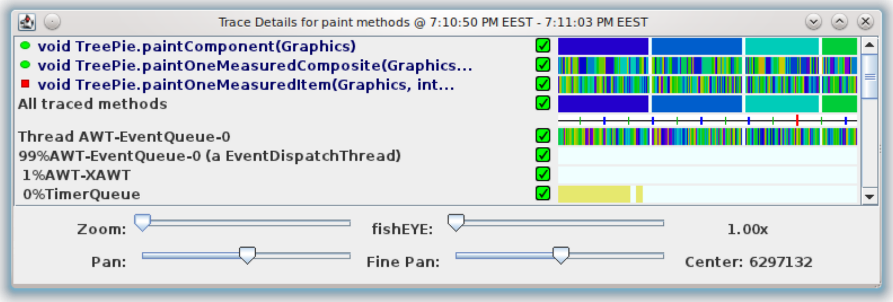
- **Details** button opens a Trace Details View showing methods traced and calls over time (see section 3).

- **GUI** button opens GUI playback controls. This button is only available if a GUI component was selected to be recorded (see section 4).

- **X-delete-X** button deletes this trace session in order to recoup memory to use in new trace sessions.

4.3 Step 3: Inspect (Using Trace and Debugger Views)

In fig.4.1, step 3 is "inspect trace data". Trace data is program state data held in values of variables, and include user-defined variables and contract status flags (Verifier data) in VERDICTS.


Trace Details for paint methods @ 7:10:50 PM EEST - 7:11:03 PM EEST

void TreePie.paintComponent(Graphics)
void TreePie.paintOneMeasuredComposite(Graphics...
void TreePie.paintOneMeasuredItem(Graphics, int...
All traced methods

Thread AWT-EventQueue-0
99%AWT-EventQueue-0 (a EventDispatchThread)
1%AWT-XAWT
0%TimerQueue

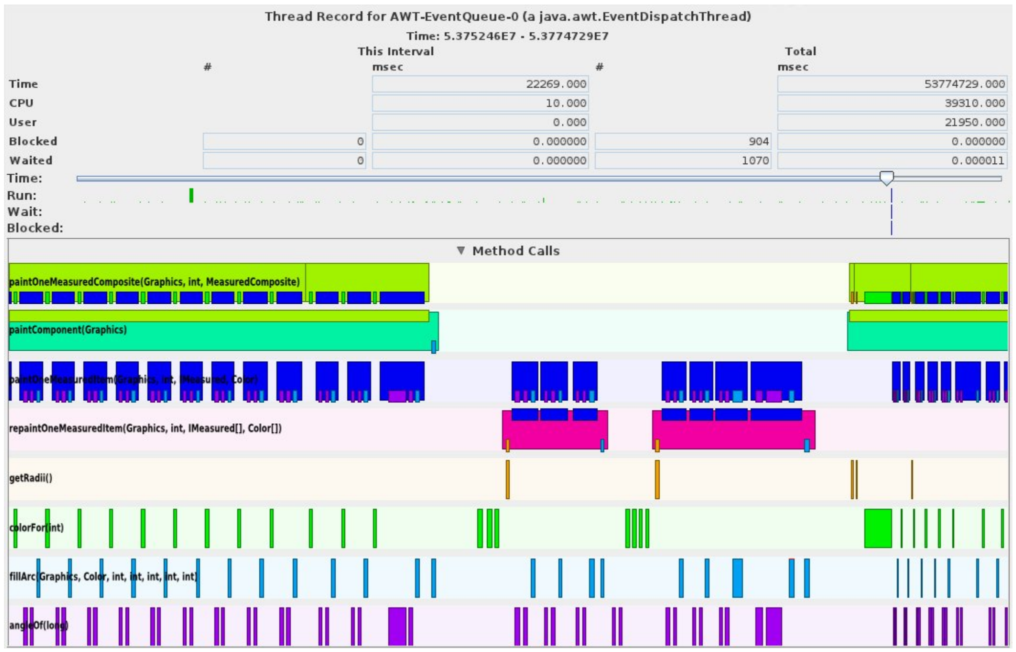
Zoom: fishEYE: 1.00x
Pan: Fine Pan: Center: 6297132

Figure 4.5: VERDICTS Trace Details View shows methods, execution time, and calls, as well as thread status.

As mentioned before, VERDICTS uses three types of views:

- *Trace views*: Show method calls over time, possibly visually marking caller-callee relationships.
- *Standard debugger views* (Individual views): Standard debugger views, showing values of variables at method call entry and exit.
- *Aggregate views*: Visualizations of data gathered across multiple method calls, to see patterns of behavior.

This section will cover first two types of views, as these are basic means for inspecting tracing session, program state and contract status flags in VERDICTS. Even though trace views and aggregate views both show multiple method calls, trace views only show when method calls start and end whereas aggregate views process accumulated data (values of variables) from multiple method calls.


Thread Record for AWT-EventQueue-0 (a java.awt.EventDispatchThread)

Time: 5.375246E7 - 5.3774729E7

	This Interval #	This Interval msec	Total #	Total msec
Time		22269.000		53774729.000
CPU		10.000		39310.000
User		0.000		21950.000
Blocked	0	0.000000	904	0.000000
Waited	0	0.000000	1070	0.000011

Time:
Run:
Wait:
Blocked:
Method Calls
paintOneMeasuredComposite(Graphics, int, MeasuredComposite)
paintComponent(Graphics)
paintOneMeasuredItem(Graphics, int, IMeasured, Color)
repaintOneMeasuredItem(Graphics, int, IMeasured[], Color[])
getRadii()
colorFor(int)
fillArc(Graphics, Color, int, int, int, int, int)
angleOf(long)

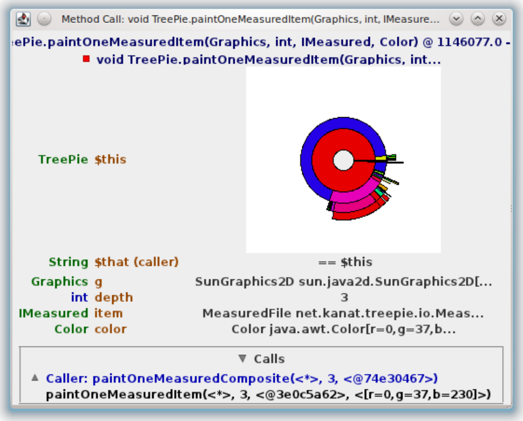
Figure 4.6: VERDICTS Thread View shows thread information and box-inside-box views of method calls, where the smaller boxes show the calls made during this call.

There are two types of trace views in VERDICTS:

- **Trace Details View** (fig.4.5) shows method-versus-time chart with boxes to show method calls. Also shows threads and thread states over time.

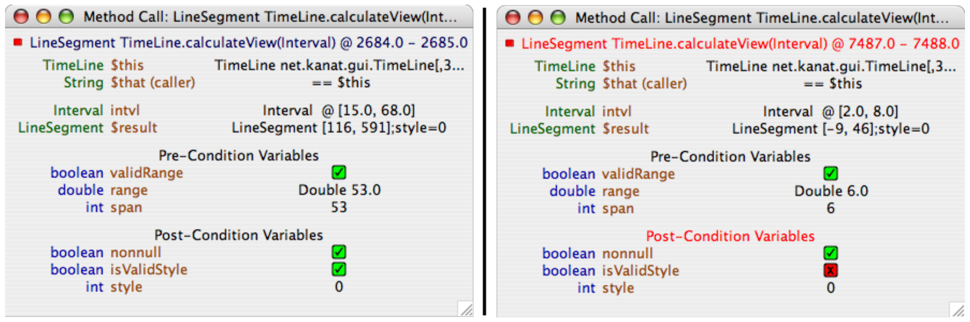
- **Thread View** (fig.4.6) is similar to Trace Details View, but shows one thread instead of all threads. Thread View also uses box-inside-box views to visually signal calls made from this call (method calls that are below in the method call hierarchy).

Trace views only show method call start and end times for all methods and calls that are traced. After contracts are added, failed calls (calls with failed contracts) are highlighted in these views in red. Other than this case, trace views do not show or reveal any data collected during tracing, whereas standard debugger views (individual method call views) we will see next and the aggregate visualizations we will see in the next section both depend heavily on the data collected during tracing session. We will make a few observations on this data, which will also better prepare us for our later discussion of adding contracts in VERDICTS.



```
Method Call: void TreePie.paintOneMeasuredItem(Graphics, int, IMeasure...
ePie.paintOneMeasuredItem(Graphics, int, IMeasured, Color) @ 1146077.0 -
■ void TreePie.paintOneMeasuredItem(Graphics, int...

TreePie $this

     String $that (caller)          == $this
   Graphics g                 SunGraphics2D sun.java2d.SunGraphics2D[...
        int depth                           3
  IMeasured item              MeasuredFile net.kanat.treepie.io.Meas...
      Color color                Color java.awt.Color[r=0,g=37,b...

                              ▼ Calls
▲ Caller: paintOneMeasuredComposite(<*>, 3, <@74e30467>)
  paintOneMeasuredItem(<*>, 3, <@3e0c5a62>, <[r=0,g=37,b=230]>)
```

Figure 4.7: VERDICTS Method Call View shows the interface, variables (including arguments and return value), callers and called methods (other Method Call windows). In this call, the "$this" object happens to be a GUI component, and is therefore shown visually.



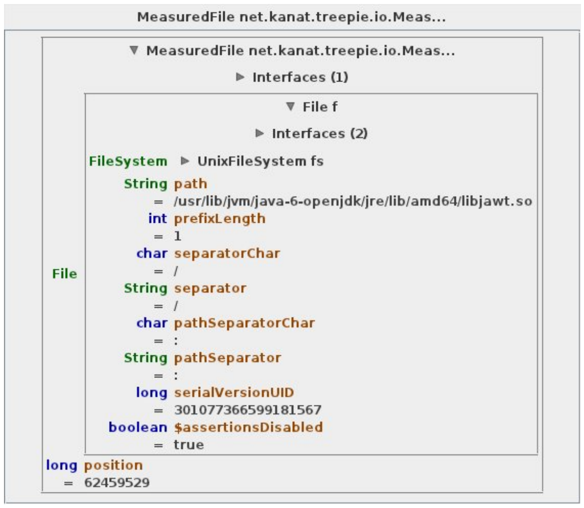
```
Method Call: LineSegment TimeLine.calculateView(Int...
■ LineSegment TimeLine.calculateView(Interval) @ 2684.0 – 2685.0
    TimeLine $this            TimeLine net.kanat.gui.TimeLine[,3...
      String $that (caller)           == $this

    Interval intvl                 Interval  @ [15.0, 68.0]
 LineSegment $result          LineSegment [116, 591];style=0

                 Pre-Condition Variables
     boolean validRange                  ✓
      double range                  Double 53.0
         int span                       53

                 Post-Condition Variables
     boolean nonnull                     ✓
     boolean isValidStyle                ✓
         int style                       0
```

```
Method Call: LineSegment TimeLine.calculateView(Int...
■ LineSegment TimeLine.calculateView(Interval) @ 7487.0 – 7488.0
    TimeLine $this            TimeLine net.kanat.gui.TimeLine[,3...
      String $that (caller)           == $this

    Interval intvl                 Interval  @ [2.0, 8.0]
 LineSegment $result          LineSegment [-9, 46];style=0

                 Pre-Condition Variables
     boolean validRange                  ✓
      double range                  Double 6.0
         int span                        6

                 Post-Condition Variables
     boolean nonnull                     ✓
     boolean isValidStyle                ✗
         int style                       0
```

Figure 4.8: VERDICTS Method Call View after user has defined some (pre- & post-condition) variables and has made some contract assertions. One post-condition assertion and therefore the call has failed on the right. This failed call and method would be highlighted in red in trace and thread views as well.

There are two types of views to inspect program status for an individual method call in VERDICTS:

- **Method Call View** (fig.4.7, fig.4.8) displays the values of arguments to the method call, the “$this” object, the caller object if known (“$that”), and the return value. Simple values are directly visible, GUI components are shown visually, and objects can



```
MeasuredFile net.kanat.treepie.io.Meas...
   ▼ MeasuredFile net.kanat.treepie.io.Meas...
              ▶ Interfaces (1)
                     ▼ File f
                  ▶ Interfaces (2)
      FileSystem  ▶ UnixFileSystem fs
          String path
               = /usr/lib/jvm/java-6-openjdk/jre/lib/amd64/libjawt.so
             int prefixLength
               = 1
            char separatorChar
File           = /
          String separator
               = /
            char pathSeparatorChar
               = :
          String pathSeparator
               = :
            long serialVersionUID
               = 301077366599181567
         boolean $assertionsDisabled
               = true
long position
  = 62459529
```

Figure 4.9: VERDICTS Object Viewer shows names, types and values of fields for an object, using collapsible boxes.

be clicked on to be examined further. If the user defines some variables and makes contract assertions, this view will also show the values and status for these, as seen in fig.4.8. A single failed assertion fails the call and the method, and this is highlighted in red in every view that shows the method or the call.

- **ObjectViewer** (fig.4.9) allows viewing fields of individual objects using collapsible GUI components.

Whenever an object appears in a Method Call View or as value of a field of another object in the ObjectViewer, it is depicted with a hyperlink (or an image for GUI component objects) that can be clicked. Clicking this link (or image) pops open an ObjectViewer for that object. In Java, arrays are special types of objects. An array-value in a field is handled specially in ObjectViewer as well: Rather than generate a generic object hyperlink, ObjectViewer shows arrays as expandable/collapsible lists of elements.

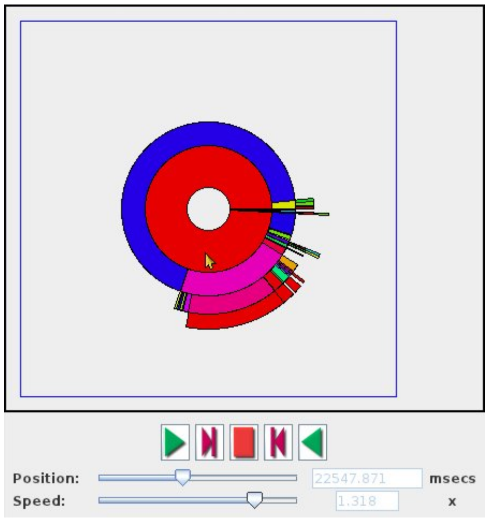


Figure 4.10: VERDICTS GUI Recorder View allows forward/reverse playback of program execution during tracing session, through DVD player type controls (plus fine-control for speed) for a GUI component's recording.

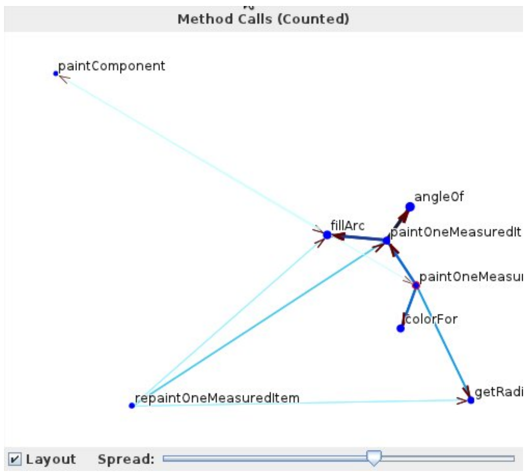


Figure 4.11: VERDICTS Methods Graph View shows degree of association between methods (according to number of times one method calls the other) using a force-based layout algorithm.

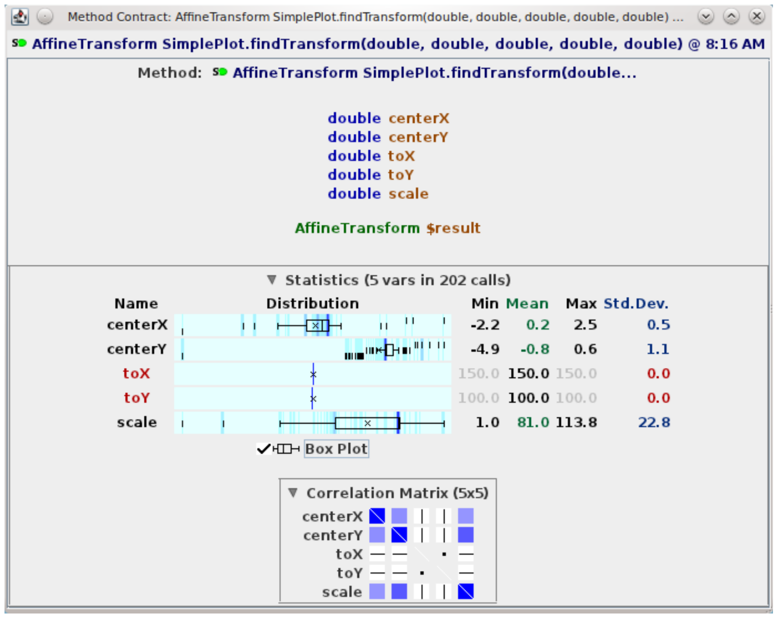
4.4 Step 4. View The Aggregate Views (Visualizations):

As stated before, aggregate views are visualizations of data gathered across multiple method calls. These help us see patterns of behavior over time. Such visualizations are not available in standard debuggers as debuggers do not record past program state, and are therefore woefully inadequate in helping us see patterns in behavior.

There are four types of aggregate views in VERDICTS:

- **GUI Recorder** (fig.4.10) plays back (forward/reverse, with speed control) a GUI component that was recorded earlier during a tracing session. During playback, point of execution time is also highlighted in all open trace views (trace details view and thread views) for this tracing session.

- **Methods Graph** (fig.4.11) uses a force-based layout algorithm [2] on a directional graph of methods. The attraction force betwen the method nodes depends on the number of times one method calls the other. Strongly associated methods have more calls, and pull each other in more. In steady-state, such methods appear closer unless they are pushed apart by repelling forces of their neighboring nodes.



Method Contract: AffineTransform SimplePlot.findTransform(double, double, double, double, double) ...

AffineTransform SimplePlot.findTransform(double, double, double, double, double) @ 8:16 AM

Method: AffineTransform SimplePlot.findTransform(double...

```
double centerX
double centerY
double toX
double toY
double scale

AffineTransform $result
```

▼ Statistics (5 vars in 202 calls)

Name	Distribution	Min	Mean	Max	Std.Dev.
centerX		-2.2	0.2	2.5	0.5
centerY		-4.9	-0.8	0.6	1.1
toX		150.0	150.0	150.0	0.0
toY		100.0	100.0	100.0	0.0
scale		1.0	81.0	113.8	22.8

✓ Box Plot

▼ Correlation Matrix (5x5)

Figure 4.12: VERDICTS Statistics View is part of Method Contract View that we will see in the next section, and contains Boxplot and Correlation Matrix described earlier, in ch.3. Boxplots depict spread of individual variables. Correlation matrix shows degree of linear correlation between variables, and highlights some trivial relationships.

[2] This is equivalent to a physical system of nodes (here, methods) with the same electrical charge (naturally repelling each other), with each edge corresponding to a spring (of equilibrium length 0) where the spring constant depends on some property of the pairs of nodes (here, the number of calls between methods).

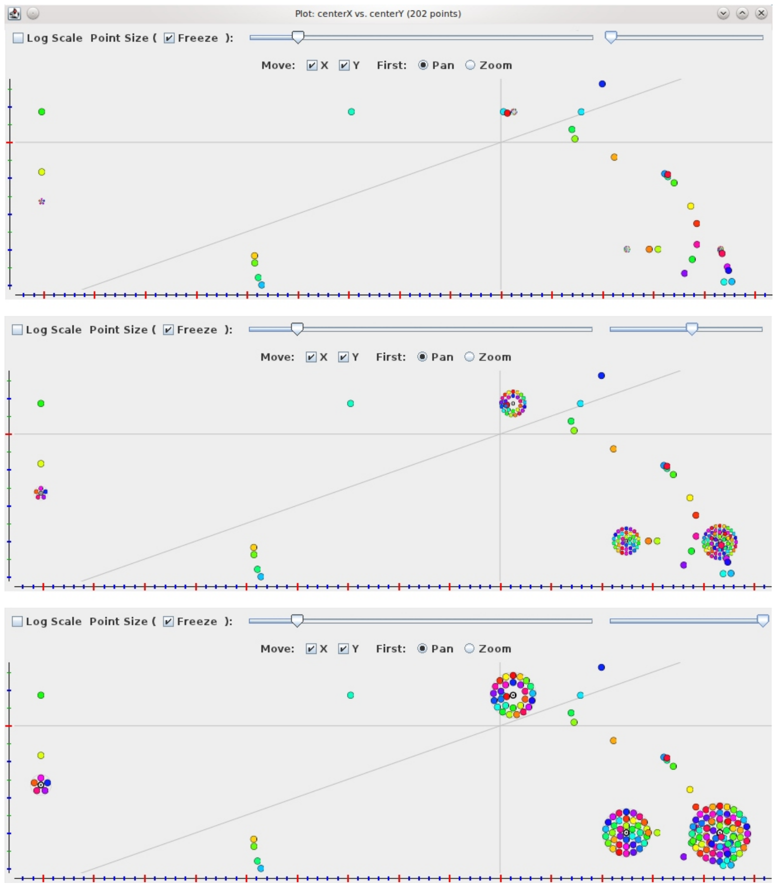


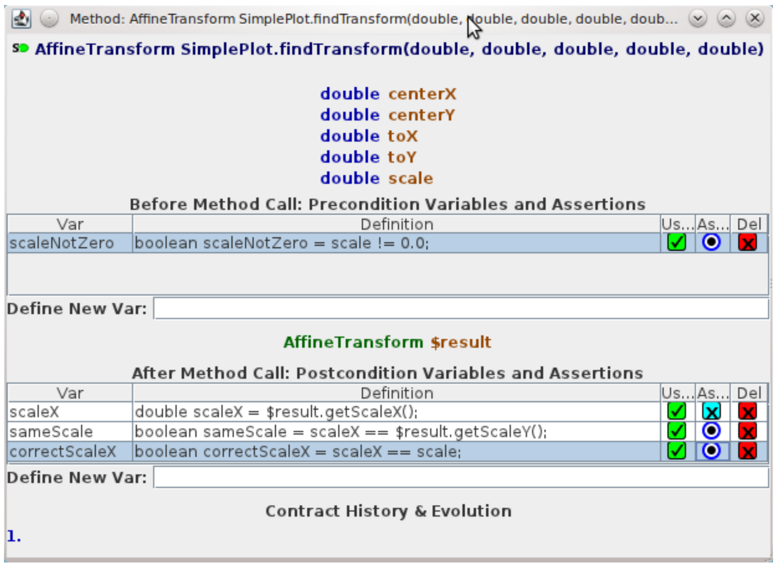
Figure 4.13: A VERDICTS X-Y Plot shows relationship between any two traced (possibly user-defined) variables. The plot above is for variables centerX and centerY. Repeated values of (x,y) across calls can be expanded into packed circles as seen in these plots. Clicking on a circle opens up the Method Call View for that call.

- **Statistics View** (fig.4.12 and fig.4.16) is part of the **Method Contract View** that will be explained in the next section. Before a contract is defined, this view only contains method interface variables as seen in fig.4.12. Statistics View depicts population statistics for values of local & user-defined variables gathered across method calls of one method. This view shows some statistics numerically for each variable, and contains two types of VERDICTS visualizations that were explained in some detail earlier, in chapter 3:
  - **Boxplot**: A horizontal boxplot view on top of a smoothed population density background.

  - **Correlation Matrix**: A view that uses color to depict Pearson's correlation coefficients between pairs of variables, and also checks for and reveals some trivial relationships (x=y, x constant, y constant, both x & y constant).

- **X-Y Plots** (fig.4.13) plot one variable against another for all calls of one method. This allows seeing patterns between related variables.

4.5 Step 5. Discover/Improve Contract.



Method: AffineTransform SimplePlot.findTransform(double, double, double, double, doub...

AffineTransform SimplePlot.findTransform(double, double, double, double, double)

double centerX
double centerY
double toX
double toY
double scale

Before Method Call: Precondition Variables and Assertions

Var	Definition	Us...	As...	Del
scaleNotZero	boolean scaleNotZero = scale != 0.0;	✓	●	✗

Define New Var:

AffineTransform $result

After Method Call: Postcondition Variables and Assertions

Var	Definition	Us...	As...	Del
scaleX	double scaleX = $result.getScaleX();	✓	✗	✗
sameScale	boolean sameScale = scaleX == $result.getScaleY();	✓	●	✗
correctScaleX	boolean correctScaleX = scaleX == scale;	✓	●	✗

Define New Var:

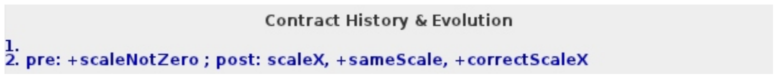
Contract History & Evolution

1.

Figure 4.14: VERDICTS Method View shows method signature and allows user to enter and see the current revision of the contract (variables & assertions).

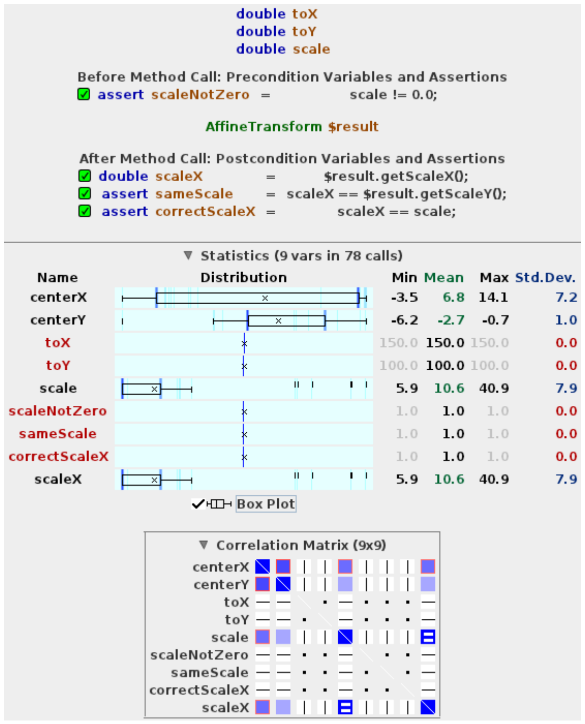
For software testing, the VERDICTS user knows what is expected of a method, and can codify this in the form of a method contract. For software analysis, VERDICTS user must perform various inspection and visualization tasks before creating a hypothesis of method requirements. Such hypotheses can then be converted to candidate method contracts.

Method contracts are created on the Method View (see fig.4.14) that shows method signature and the current version of the contract, allowing editing such contract. Contract is made up of user-defined variables. Boolean variables can be marked as "asserted", which promotes those variables to be part of the method contract's requirements. If an asserted variable is false in a method call, that call and the method (or method contract) have failed. Other helper variables can also be defined to make the assertion expressions easier to follow.


Contract History & Evolution

1.
2. pre: +scaleNotZero ; post: scaleX, +sameScale, +correctScaleX

Figure 4.15: VERDICTS Method Contract History at the bottom of Method View shows revisions of contracts (user-defined variables and their definitions). Revisions are immutable copies of contracts that are automatically generated when a tracing session is run after a contract is defined/modified. A "+" marks an assertion.


```
double toX
double toY
double scale

Before Method Call: Precondition Variables and Assertions
✓ assert scaleNotZero  =                scale != 0.0;

AffineTransform $result

After Method Call: Postcondition Variables and Assertions
✓ double scaleX        =         $result.getScaleX();
✓ assert sameScale     =  scaleX == $result.getScaleY();
✓ assert correctScaleX =            scaleX == scale;
```

▼ Statistics (9 vars in 78 calls)

Name	Distribution	Min	Mean	Max	Std.Dev.
centerX		-3.5	6.8	14.1	7.2
centerY		-6.2	-2.7	-0.7	1.0
toX		150.0	150.0	150.0	0.0
toY		100.0	100.0	100.0	0.0
scale		5.9	10.6	40.9	7.9
scaleNotZero		1.0	1.0	1.0	0.0
sameScale		1.0	1.0	1.0	0.0
correctScaleX		1.0	1.0	1.0	0.0
scaleX		5.9	10.6	40.9	7.9

Figure 4.16: VERDICTS Method Contract View. This is the same combined view with statistics, seen before in fig.4.12. Now, the contract is a real contract, with user-defined variables and assertions.

A contract created or modified and used in a tracing session is saved as a frozen contract, a revision in this method's contract history, as seen in fig.4.15. In this figure, revision 1 is empty contract, with only method's own interface variables, with no user-defined variables or assertions. Revision 2 has one pre-condition variable and three post-condition variables, and except for scaleX, all other variables are boolean variables that are asserted to be true. Clicking on this revision opens up the Method Contract View seen in fig.4.16.

We have seen Method Contract View for empty contract earlier, in fig.4.12 while examining the Statistics View, with boxplot and correlation matrix. One difference is that, now, we have user-defined variables as well (everything below "scale" in fig.4.16). Boxplot and univariate statistics highlight the constant-valued "variables" toX, toY, scaleNotZero, sameScale and correctScaleX. For these variables, the names and standard deviations are highlighted in red whereas minimum and maximum values (which equal to mean) are faded out in gray. The last three constant-valued variables are actually our three asserted boolean variables, and it is desirable to have them always return true as they do here (floating point value 1.0 represents true in this case).

The correlation matrix in this figure also shows:

- strength of correlation with darker and more saturated blue.
- negative correlation with a red-line around the blue box
- constant-valued variables (horizontal or vertical line, or a single point when both X and Y are constant)
- the diagonal (around which correlation matrix is always symmetric)
- cases outside the diagonal where X == Y at all times, with an "=" symbol"

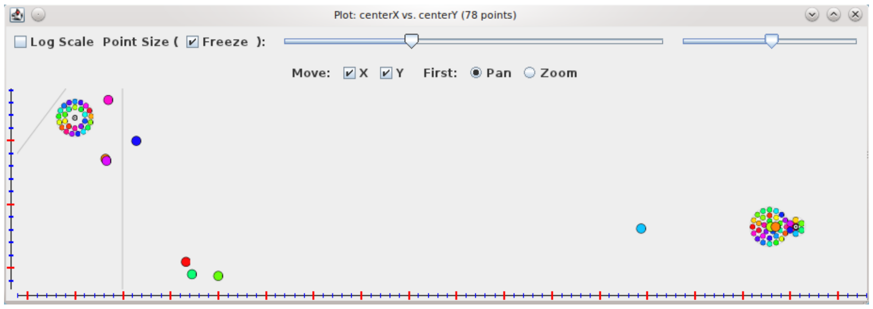


Figure 4.17: VERDICTS X-Y Plot after declaring our method contract. Compare with fig.4.13.

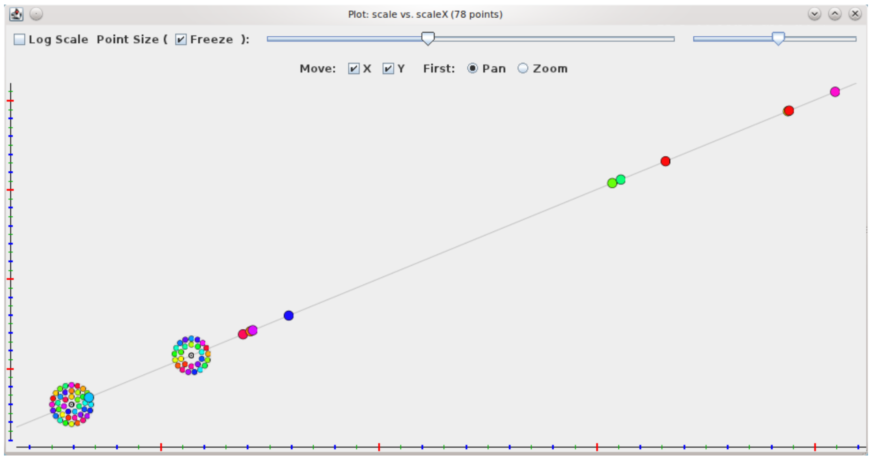


Figure 4.18: VERDICTS X-Y Plot using a user-defined variable from method contract (scaleX), showing a trivial relationship, scale == scaleX.

As before, user can click on any box in the correlation matrix to see a plot of values. Fig.4.17 shows plot of centerX vs. centerY, and fig.4.18 plots scale against scaleX. From correlation matrix view where there is an "=" symbol, we already know that scale = scaleX.

The many controls and interfaces VERDICTS provides can be used in different combinations for various analysis and testing scenarios.

4.6 User-Defined Variables: Observables and Contract Assertions

Compared to debugging, which allows access to full internal state of the program, VERDICTS allows a much more limited view. But debuggers do not record program state data over its execution, whereas VERDICTS does. This allows going backwards in execution, and aggregate analysis of such data.

As a program's execution goes through a large number of debug step points (millions per second in today's computer speeds), keeping all program state information causes a serious execution time penalty; even the most efficient algorithms researched report multiple orders of magnitude difference in software execution speed [58] [48].

VERDICTS allows recording any amount of data over program state by dynamically changing decisions over what to record while evolving hypotheses about software requirements and behavior. This is achieved by user-defined variables ("observables") whose definitions can change without the need to stop and recompile the target program.

The default level of recording is at function interactions rather than individual statements: Inputs to and outputs from a function are recorded for all functions of interest.

Compared to traditional tracing, VERDICTS Tracer collects much more data, as it records all interface variables used in inter-method communication.

At method entry, VERDICTS records:

- values of all arguments
- this object reference if applicable (held in special variable called "$this" in VERDICTS)
- the caller object if known ("$that" in VERDICTS; holds reference to the object whose method called this method),

At method exit, VERDICTS records the return value, if one exists ("$result" in VERDICTS).

Note that as we mentioned before Java does not allow any object to be copied, and all objects are passed by reference. By default, VERDICTS makes copies of primitive values, but only records the references for reference types. Note that in the special case of Strings, which are immutable, it is sufficient to only keep references as the object referred to (the string) is unmodifiable.

Beyond these standard variables which are recorded during method entry and exit, the user can also define variables to be evaluated and recorded at method entry and exit points. In VERDICTS, we think of such variables as probes into the otherwise closed program's internal state, and these variables are called user-defined "observables". Even though such observables were only mentioned as part of contracts earlier, it is quite possible that many cycles of creating observables and running VERDICTS Tracer-Verifier may pass before observables evolve and patterns between observables are noticed and coded into contract assertions.

In the following subsections, we will examine different types of observables that can be used in VERDICTS.

4.6.1 Object Clone (Copy): Advantages, Disadvantages, Finding a Good Compromise

For any array or other type of object in Java, recording just the reference may not be enough as the object may have changed since its reference was recorded. If the language allows copying/cloning the object in question, the user can define a variable that holds a clone of such an object. Shallow or deep copies can be created if the objects to clone are instances of classes that allow these.

Consider a merge method used in mergesort:

```
/** Merges sorted (in nondecreasing order) subsequences
 *  source[i0..i1-1] and source[i1..i2-1] into dest[i0..i2-1].
 *  After return, dest[i0..i2-1] is sorted in nondecreasing order.
 */
void merge(int[] source, int[] dest, int i0, int i1, int i2)
```

Here, VERDICTS will only record the references of the arrays source and dest. Mergesort can be efficiently implemented by swapping source and dest array references at each call depth. In this way, the states of these arrays will continually change during mergesort. After many calls to merge, it is not possible to understand an earlier call as we did not save the state of the array at the time of the call. To store more data, the user can define two observables which copy (clone) the source and dest arrays:

```
Precondition declarations (execute at method entry):
    int[] src = (int[])source.clone();
  Postcondition declarations (execute at method exit):
    int[] dst = (int[])dest.clone();
```

Now, as these user-defined expressions will be evaluated at the time of method call entry and exit, they will make full copies of arrays of interest, and we can examine earlier calls to merge without worrying about loss of program state.

But there is a problem with creating too many clones. If mergesort was used for an array of 1024 integers, and merge was used for merging single elements up to full array, then this merge function would be called 512 times to merge 1-element arrays, 256 times to merge 2-elements arrays... for a total of 512+256+128+...1 = 1023 times. For each call, we are making clones of two 1024-element arrays, so we use 2046 arrays of size 1024 integers to track one call to mergesort, whereas mergesort itself needs just one more 1024-element array other than the original array to be sorted. Compared to recording two reference values per call to merge (and assuming integers and references are same size), this represents an overhead of 1024 times, which is three orders of magnitude larger. Obviously this much array cloning would also cause significant speed overhead as well. Also, this memory usage and speed complexity is O(N^2) for array size N, whereas mergesort has speed complexity of O(N x log(N)) and memory complexity of O(N), so the overhead is significantly higher for larger arrays.

But as VERDICTS allows the user to decide what level of data recording overhead and speed overhead is needed and is sufficient, we can find a more efficient compromise between storing sufficient state and overheads (both memory and speed).

What would be better in the case of merge is to assume that merge does not corrupt anything beyond the range it claims to change, specifically, dest[i0..i2-1]. In that case, the only data that needs to be recorded are source[i0..i2-1] and dest[i0..i2-1]:

```
  Precondition declarations:
    int size = i2 - i0;  // number of elements
    if (size < 0) size = 0;  // truncate to 0 if needed
    int[] src = new int[size];
    if (size != 0) System.arraycopy(source, i0, src, 0, size);
  Postcondition declarations:
    int[] dst = new int[size];
    if (size != 0) System.arraycopy(dest, i0, dst, 0, size);
```

In this case, the previous situation would still require 2046 arrays to be recorded, but most arrays will be much smaller than 1024 elements. The bottom 512 calls to merge would be merging 1-element arrays into 2-element arrays, so both src and dst would be 2-element arrays. Ignoring the array object overhead, the memory needed would be for 512*4 + 256*8 + ... 1*2048 integers = 10*2048 = 20,480 integers instead of for 512*2048 + 256*2048 + ... 1*2048 = 1023*2048 = 2,095,104 integers.

In this case, the memory complexity is O(N x log(N)), and speed complexity is the same if we ignore the array creation overheads.

4.6.2 Recording Partial Object State: Fields and Methods

For some large objects, an alternative to full cloning is to record partial object state. Often only a few fields of an object are of interest, and recording those fields may suffice to be able to understand the behavior of the method. Consider a method that adds a person to a hashtable called phoneBook:

```
void addPerson(Hashtable<String,String> phoneBook,
               String name, String phone);
```

Even though the phoneBook may have potentially millions of entries and other object state, we are unlikely to be interested in making copies of the whole phoneBook or any significant portion of its state.

To understand the behavior of this method, we may be interested in whether the name already existed in the phoneBook as a key before this call, what phone number or string it mapped to if it existed, whether the name exists upon exit from method and what phone number or string it has at that time. Here's how we could achieve this:

```
Precondition declarations:
  String phone1_pre = phoneBook.get(name); // may be null
Postcondition declarations:
  String phone1 = phoneBook.get(name);
  boolean correct = phone.equals(phone1);
  // more strict == comparison (phone == phone1) may
  // be required as well
```

Passing a hash table to a public method is not a good encapsulation practice. A better approach is to have phoneBook as a field of the "this" object. Consider a Java class, PhoneBook:

```
public class PhoneBook {
  private Hashtable<String,String> phoneBook;
  public void clear() { ... }
  public void addPerson(String name, String phone) { ... }
}
```

In this case, the same exact user-defined (precondition and postcondition) variable declarations above would still work, to record partial object state information for a field of "this" object.

A postcondition for the clear() method may be:

```
Postcondition declarations:
  int size = phoneBook.size();
  boolean isEmpty = size == 0;
```

Here recording the size value as well could help us diagnose faulty behavior patterns when there are problems.

4.6.3 Recording Partial Object State: Properties, More Complex Processing

In the previous section we looked at simple object state that is held in fields and getter methods. Getter methods are side-effect-free methods that return some information about object state that is held in its fields. Beyond such basic partial state data, we can also gather and process much more information about the object, accumulated over multiple accesses to object internal state.

Looking back at the merge(...) example we saw earlier, if we only want to know when or if a merge fails without knowing specifics about array elements, we could merely record some checksums that are somewhat orthogonal to observable features of the function. In the case of sorted arrays, developer can examine array for sortedness by looking at elements of dest[i0..i2-1] and seeing they are sorted. But this does not make sure that the sorted array actually contains the same elements are the two subarrays that merge is required to merge. The expensive solution is to check that output is a permutation of the input (same bag/multiset of values). But this is costly to check. For lower memory and processing costs, we could use a simple checksum approach, and compare total of elements in source[i0..i2-1] and compare this with totals of dest[i0..i2-1] at method exit:

```
Precondition declarations:
  int sumSrc = 0;
  for(int i = i0; i < i2; i++)
    sumSrc += source[i];
Postcondition declarations:
  int sumDst = 0;
  for(int i = i0; i < i2; i++)
    sumDst += dest[i];
  boolean sameSum = sumSrc == sumDst;
  // and assert sameSum
```

Equivalently, we can use the predefined "$sum" function in VERDICTS:

```
Precondition declarations:
  int sumSrc = $sum(source, i0, i2);
Postcondition declarations:
  int sumDst = $sum(dest, i0, i2);
```

As with parity checking, a single-element error will always cause checksum test to fail, and as checksum holds much more information than a single bit, it is also unlikely for multiple errors to cancel each other to give the same checksum.

If instead of this checksum, we want to find out is whether dest[i0..i2-1] is sorted at method exit, we can examine this with:

```
Postcondition declarations:
  boolean sorted = true;
  for(int i = i0; i < i2 - 1 && sorted; i++)
    if (dest[i] > dest[i+1])
      sorted = false;
  // and assert sorted
```

Equivalently, we can use the predefined "$sorted" function of VERDICTS:

```
Postcondition declarations:
  boolean sorted = $sorted(dest, i0, i2);
```

The "sameSum" and "sorted" flags can both be calculated and asserted. As these look at orthogonal features of the merge function, combining these assertions will yield much better results compared to using either assertion.

4.6.4 Print Statements, Logging

VERDICTS also allows any existing logging infrastructure to be used as well. VERDICTS interception + evaluation allows calling a logging function upon a method entry or exit point, possibly based on some condition about the call. Any method call information can be gathered and recorded with this log. As opposed to standard approaches, this allows logging to be dynamically added to any program running under VERDICTS without stopping or recompiling it.

Even without a logging system in place, with VERDICTS, print statements can be quickly added to or removed from method entry and exit points without the need for recompilation of the target program. As we have seen in chapter 2, print statements can reveal much more information about a program compared to standard debuggers that do not record history.

For example, instead of recording all array values during merge, if we notice that it is implemented incorrectly, we can add print statements to print the array-to-be-sorted and returned supposedly sorted array at each call. Alternatively, these values could be printed only when the returned array is not sorted even though the input arrays were sorted. This is very similar to the print statements used for quicksort function example in chapter 2.

4.6.5 Patches (Throw-Away Quick Fixes)

As Java does not declare or track whether a method called may change an object's state, VERDICTS cannot promise to be side-effect-free either. Even though most user-defined observables and contracts should be side-effect-free, there are also cases when VERDICTS can be intentionally used to modify program behavior.

If a method examined with VERDICTS is found to be buggy in a certain way, it may be possible to circumvent (have a "quick fix" for) the faulty behavior. Such quick fixes are called "patches" and are not intended to be kept in place as originally created. If a patch works, it is often replaced with proper rewrite of the method that fixes the same fault properly, without extraneous unnecessary code.

Still, a patch can be quite useful as it allows us to continue testing this and other parts of the system. With a patch in place, we can discover whether:

- our hypotheses about the mechanism of failure is correct
- our hypothesis about how to circumvent/fix the faulty behavior is correct in the use case we are studying
- the program has other buggy methods related to this behavior (fixing one method is insufficient for examined use case)
- our fix breaks other behavior (discovered by testing other use cases)

Other than unconditional patching, we can also use "recovery assertions". A recovery assertion is a fault tolerance mechanism that not only discovers but also fixes faulty internal state. Unlike in standard assertions, there is no need to halt the program, as internal state is fixed and execution can continue.

As an example, consider an object, Y, that can be fully generated from another object, X. Y may be incrementally modified to stay in sync with X over time while X changes state. If a bug causes Y to fall out of sync with X, a recovery assertion that discovers this situation can be used to recreate Y object from scratch using X. Although this can be inefficient if done frequently, this allows program to continue to run correctly.

Recovery assertions can neutralize effects of bugs before they can manifest as failures. They can be used to recover from one error to discover if any other errors exist in the program, similar to how a compiler reports multiple errors. As they can hide the existence of errors, they must be used carefully. The user must be attentive and notice failed assertion to realize there was a bug and a faulty program state, even though recovery assertion has fixed the program state and program otherwise appears to be bug-free.

4.6.6 Contract Assertions

We have seen a number of boolean variables in previous sections for observations on method behavior. We have defined most boolean variables in a way that would make them true whenever the method behaved as per our expectations. Such booleans can be directly declared as assertions in VERDICTS. Unlike the assert keyword in various languages, VERDICTS assertions are more often like boolean variable declarations, with name and value:

```
assert sorted = $sorted(dest, i0, i2);
```

This is similar to Java language statements:

```
boolean sorted = ...
assert sorted;
```

One important distinction is that VERDICTS records failed assertions without stopping the running program. For state-corrupting bugs, this could cause many other assertions to fail. But note that VERDICTS allows experimenting with potentially incorrect assertion candidates, and an assertion may fail because assertion itself rather than the program behavior is faulty. If the user wants to stop the running program, VERDICTS can be used to add dynamic code that conditionally throws an exception. In Java, an unchecked RuntimeException such as IllegalArgumentException can be thrown at any time from any method without the need to declare that the method may throw such an exception.

In the next section, we will look at various types of assertions we may be interested in declaring in VERDICTS.

4.7 Types of Contracts and Requirements in VERDICTS

As mentioned before, VERDICTS uses DBC contracts to state requirements. If we know the requirements, failure of contract marks software faulty behavior. In this situation, the correct response is to try to understand details of software behavior to find where and how software fails.

A second situation arises with software in widespread use, without documentation of requirements. To discover requirements, we must guess requirements, create contracts that represent our hypotheses, and evolve such contracts till we discover the correct requirements. If software is generally believed to have been implemented correctly, failure of such a hypothetical requirement contract marks error in our hypotheses.

A third situation is when we are not sure if software is implemented correctly with respect to its requirements. For software without proper documentation, it is quite understandable that two developers may have informally agreed on an interface or expected behavior, but differently understood and implemented their halves around this interface. Common errors are easily discovered by testing, but some conditionally arising situations may not have been properly tested. In this case we cannot be certain that there is a uniform set of requirements that both a caller of a method and the method implementation itself agree upon. The most we can attempt to understand is the actual behavior of the method.

In this third situation, we can still create hypotheses, but this time of actual behavior rather than requirements. Behavior hypotheses can similarly be converted to contracts, tested, and evolved. The goal is to understand and state how the program actually works, whether this behavior corresponds to desired behavior or not.

In VERDICTS, therefore, we have these two main types of contracts:

- **Requirement Contracts** are based on what callers expect
- **Behavior Contracts** are based on the actual implementation

In the use case of discovering fault origin, instead of fully specifying the requirements or the current behavior, we might be interested in only specifying the one requirement that we believe is not followed, or the one buggy behavior that we believe causes program to not conform to its requirements. For example, a buggy implementation of sortAscending(int[]

ar) method may have:

- Requirement: Array ar is sorted in nondecreasing order upon exit from method
- Behavior: $ar[0]$ is never modified from entry to exit

These contracts are conflicting in general for most arrays; the only situation they will not conflict is when $ar[0]$ is always the minimum element in array when the method is called.

Failure of a requirement contract would mark faulty requirement hypotheses (callers did not require this, in this case). Failure of a behavior contract would mark faulty understanding of the method implementation; this method does not behave in this way. In the above example, "$ar[0]$ is never modified" is a specific behavior that contrasts required behavior. Failure of this contract does not mark faulty implementation. Actually, nonfailure of this contract with a wide range of inputs suggests faulty implementation, and a correct hypothesis for mechanism of failure. A sorting method does not always have to modify first element of the array, but it also should not be implemented so that it never modifies the first element. If this contract passes our tests, we have discovered a fault origin. Note that there may be other faults in implementation as well.

Beyond full and partial functional requirements contracts and behavior (actual implementation) contracts, we can also use fuzzy contracts that are expected to often but not necessarily always hold. These can be used to visually highlight exceptions to observed pattern of behavior, and could be used to discover behavior trends.

Table 4.1 compares these various types of contracts, and table 4.2 shows examples of contracts for the sortAscending(int[] ar) method mentioned above. In these tables, R represents requirements, P represents behavior. R may allow various decompositions into partial requirements R$_i$, and P may similarly allow various decompositions into partial behavior specifications P$_j$. A defect causes program behavior to contradict a requirement; $\exists i, j : P_j \implies \neg R_i$ (equivalently, $\exists i, j : R_i \implies \neg P_j$, and also, $\exists i, j : \neg(P_j \wedge R_i)$).

Table 4.1: Types of contracts that can be used in VERDICTS for analysis and testing

Contract Type	Predicates	Failure Marks
Requirements	R = $R_1 \wedge R_2 \wedge \dots$ (there may be multiple decompositions of R with different R_i)	faulty implementation or faulty requirement
Behavior (actual implementation)	P = $R_1 \wedge R_2 \wedge \dots$	wrong comprehension (of implementation)
Failure mechanism (pro)	P_j (with $\exists i : R_i \implies \neg P_j$)	wrong comprehension
Failure mechanism (con) partial requirement	R_i (with $\exists j : R_i \implies \neg P_j$)	correct comprehension, and a case of failure
General pattern; fuzzy	S (where S "often" holds)	exception to the pattern

Table 4.2: Examples for types of contracts that can be used in VERDICTS for analysis and testing, for a sortAscending method given integer array ar with length n.

Contract Type	Example for sortAscending(int[] ar)
R: Requirements	R_1: Array ar is sorted on exit i=1..n-1: ar[i-1] <= ar[i] R_2: ar contains same elements (is permuted; no is element lost)
P: Behavior (actual implementation)	Can be very specific (P_i: one test case), very general (P_1: ar is sorted on exit), or inbetween (P_1: n < 4 $\implies$ ar is sorted)
P_j: Failure mechanism (pro)	ar[0] == 0 on exit (contradicts R_1, R_2)
R_i: Failure mechanism (con) (a partial requirement)	for n > 1, ar[0] <= ar[1] (hypothesis: ar[0] is not sorted, so this contract will sometimes fail)
S: General pattern; fuzzy	ar[0] > 0 (observed usage pattern)

4.8 Review: How VERDICTS Supports Exploratory Contracts

Let us review how various features of VERDICTS support discovering, creating and using exploratory contracts. User can create requirements contracts as well as behavior contracts. VERDICTS supports:

- Testing and debugging (user is trusted, program is suspect/faulty):

  - by creating requirement contracts to test program against
  - by fixing and evolving such contracts
  - by testing hypotheses of program fault mechanism by applying temporary patch code through VERDICTS

- Analysis (program is trusted, user's hypotheses and contracts may be faulty):
  - by creating behavior contracts
  - by guessing, testing and evolving "candidate" requirement contracts

The three primary components of VERDICTS are designed around contract evolution and testing (see section 4.1):

- Tracer:
  - Records data beyond standard tracers, to observe historical program state data
  - Intercepts program to allow dynamic addition and modification of contracts
- Verifier:
  - Records user-defined data
  - Evaluates contracts
- Visualizer:
  - Helps user see patterns of program behavior

As mentioned above in sections 4.1 and 4.3, there are three types of views:

- Trace views: Help user notice patterns of method calls
- Standard debugger views: Help user examine a single method call and the objects involved in detail

- Aggregate views: Help user notice general patterns of behavior through relationships between data collected

VERDICTS can help user notice patterns in program behavior and state, through its various aggregate views (see section 4.3) of recorded variables that collectively make up part of program state that the user is interested in:

- GUI Recorder: Allows playback of any recorded target program GUI behavior execution-time synchronized with other views.
- Methods Graph: Shows method associations on a graph (how frequently methods call each other).
- Statistical Views:
  - BoxPlot: Shows univariate spread (quartiles, median, outliers)
  - Correlation Matrix: Shows bivariate linear relationship strengths
- X-Y Plots: Shows bivariate relationship. If a regularity is observed, this could be coded as a candidate contract assertion.

Standard debugger views, statistical views and X-Y plots depends on values of variables. Other than interface variables, VERDICTS allows definition of user-defined variables. This is also the mechanism to define contracts in VERDICTS: Contracts are made up of assertions of boolean variables. Automatic versioning of contracts (upon starting VERDICTS trace) allows seeing how variable declarations in contracts have evolved over time. VERDICTS variables support exploratory contracts by giving dynamic (without need to stop target program or recompile it) and full access to objects and the programming language. As we saw in section 4.6, this includes cloning objects, loops, throwing exceptions, print statements, and access to some useful predefined functions/macros.

These features of VERDICTS provides user with a rich environment and quick feedback when using exploratory contracts for software analysis and testing.

4.9 VERDICTS Tests

In this section, we will:

- Examine earlier tests of VERDICTS mentioned in our published work
- Show results from a new set of tests for three of the innovative features of VERDICTS, comparing VERDICTS to Eclipse IDE.

4.9.1 VERDICTS Efficiency and Earlier Tests

In [13], we ran an an earlier version of VERDICTS (that uses the same interception code as current version) on a Mac Mini with G4 1.25 GHz PPC processor and 512 MB memory. We observed that the performance degradation is not significant and the system remains interactive, even when we record 2783 method calls per second, as seen in table 4.3. Recording 2783 method calls per second caused a performance degradation of 35.5%. In comparison, on this system, moving the mouse slowly to traverse the diagonal span of the screen over two seconds caused about 30% performance degradation.

Table 4.3: VERDICTS method interception overhead

Time (sec)	Method Calls	Calls / sec	Performance Degradation (%)
20.8	11,728	564	19.2
19.9	55,385	2,783	35.5

As we mentioned before, fully recording program state by using checkpointing, as is done in reversible and bidirectional execution methods, would generate a significantly higher performance degradation. Whyline [48] reports overhead factors of 4.1 - 14.3, which are 20 to 40 times larger than VERDICTS overhead.

In [13], we also reported that trace details view allowed us to notice inefficient implementation for a component which performed correctly, in that it did not have any noticeable faults in behavior. Viewing internal state of the program allowed us to notice problems even when we were not intentionally trying to discover any bugs. Even though our program appeared fast, it was very inefficient, and would not scale up very well.

In [15], we demonstrate how we have used VERDICTS to discover fault origins in two non-vital components of VERDICTS. In those cases, the dual nature of VERDICTS allowed us to have a wider perspective and observe patterns of method calls with visual exploration, and discover the component responsible from faulty behavior by dynamically added observables and contracts.

In the first case, we noticed an unexpected pattern, quickly focused on problematic methods, created general-purpose observables and discovered fault origin in a matter of minutes. The aggregate views allowed us to notice patterns and behavior that failed to follow common patterns. This would not be possible with a debugger. Interestingly, we discovered that the fault origin was in a publicly available library rather than our own code. The standard Java library method to find current memory usage used a coarse-grain update instead of real-time update with each memory allocation. This was not at all implied in any way by the library API documentation.

In the second case, we used dynamic analysis to discover faulty method, and code review to find exact fault origin. In accordance with our observation that GUI component is not being updated properly, we added a requirement assertion that we expected to always fail. Specifically, our contract stated internal state should change with each call to a certain method. This was a "fuzzy" requirement contract; we expected it to be satisfied "most of the time" (but not always) in a correctly running program, and to always fail in our buggy program. Infrequent failure of this contract would not signify a bug. The dynamic analysis quickly showed us that we were wrong; our contract rarely failed, and our hypothesis about mechanism of failure was wrong. Quick negative feedback helped us not go down farther along the path we had expected to find the bug. A debugger can help discover when a condition is true or false, but cannot show us whether a condition is satisfied some of the time, most of the time, or rarely (and under which conditions), so it does not help with fuzzy requirements like the one we used in this case.

For more details, please see [15] and [13], both available online at [16].

4.9.2 Testing Innovative Features of VERDICTS

For these tests, we compare VERDICTS (version 1.0) to a modern Java IDE, Eclipse (version 3.7.2). Our test system is a quad-core Dell Vostro intel i5-2410 laptop running at 2.3 GHz, with Ubuntu 11.04 (with Linux kernel 2.6.38-15-generic).

We aim to prove three innovative features of VERDICTS:

- A. Integrating large amounts of program values in comprehensive but useful ways
- B. Using novel visualizations that reveal patterns in control flow and data variations
- C. Dynamically inserting probes and hypotheses about program behavior using a familiar language

As Eclipse doesn't provide visualizations, we will use all other available tools in Eclipse to accomplish the same tasks.

We ran tests on four targets:

- JUnit: JUnit 4.8.2
- Ant: Apache Ant 1.9.2
- Commons: Apache Commons Collections 3.2.1
- VERDICTS: VERDICTS 1.0

For each target, we selected two parts of the project with similar size source code. Our eight experiment targets are:

- JUnit1: org.junit.runner package [3]
- JUnit2: org.junit.runner.notification and org.junit.runner.manipulation packages
- Ant1: Java and ExecuteJava classes in org.apache.tools.ant.taskdefs package
- Ant2: org.apache.tools.ant.types.Path class and its inner class, Path.PathElement.
- Commons1: org.apache.commons.collections.PredicateUtils class
- Commons2: org.apache.commons.collections.MultiHashMap class
- VERDICTS1: ZoomPanComponent and Painter classes in net.kanat.gui package
- VERDICTS2: net.kanat.gui.XYPlot class and its two inner classes, XYPlot.Controller and XYPlot.RangePosAndNeg.

In the first test for each target project, we tested VERDICTS first, and Eclipse second. In the second test for the target project, we tested Eclipse first and VERDICTS second. These pairs of tests were not conducted right after one another, but other tests were ran in the interim to forget knowledge gleaned from one tool, to reduce bias. This has generally worked very well; most of what was learned was forgotten between tests, in part also because of VERDICTS and Eclipse having very different interfaces, and Eclipse providing only a depth-first traversal through the method calls.

Different main programs in different environments were run for these tests.

JUnit is a unit testing framework. For JUnit1, we used a simple test class with four simple methods tested (helloWorld(), hello(String), addition(int,int), gcd(int,int)). A few of our test cases are intentionally wrong, so that two of our four tests fail even though implementation is correct. JUnit2 tests were actually conducted earlier, and used an earlier

[3] This would not include any org.junit.runner.* packages such as those used in JUnit2. In Java, package name similarity doesn't provide any special access. Java packages don't have a scope hierarchy.

version of simple test class with only two simple methods tested (helloWorld and addition), where both tests pass. [4]

Ant is a portable Java build tool. Both Ant1 and Ant2 use the same set-up. The steps for build are defined in our "build.xml":

- Delete all generated (compiled) files and their directories
- Recreate directories
- Compile depended-upon project's source code (net.kanat.commons, 22 classes)
- Compile this project's source code (net.kanat.gui, 108 classes)
- Run three GUI demo programs without forking, then run them with forking (using separate Java Virtual Machine).

Apache Commons Collections comes with a full suite of unit tests. We just run all tests. This is wasteful when targeting a single class, but is a simpler and flexible setup.

For VERDICTS, we just ran demo programs, XYPlot.main(.) in VERDICTS1, and ZoomPanComponent.main(.) in VERDICTS2 tests.

In the next three subsections, we will go through each innovative feature tested, metrics gathered, and our observations.

4.9.3 A. Integrating Large Amounts Of Program Values In Comprehensive But Useful Ways

This is really about richness of outputs of VERDICTS, mainly in its visualizations. VERDICTS is:

- Comprehensive, as it doesn't only track, record and reveal a subset, but rather all calls of all methods of interest;

[4]As there weren't sufficient number of method calls to verify contracts, for the contract tests we converted a JUnit3 type unit test from Apache Commons Collections (org.apache.commons.collections.TestMapUtil) to JUnit4 type test and used that instead, while tracking and verifying same method calls as before.

- Useful, as it allows:
  - zooming in and focusing on a subset of calls to observe local behavior,
  - examining all calls of the method through univariate and bivariate statistical measures and plots.

We used four metrics:

- ST: Statements: How many distinct (non-overlapping) general statements about control flow and data variations can we make after using tool for two minutes.
- CTRL: Control flow density: For how many methods do we understand the control flow by looking at once screen. [5]
- DATA: Data density: For how many methods do we understand the input and output data by looking at one screen. [10]
- PXL: Pixel size sufficient? Is any information lost or hard to identify due to being displayed in just one pixel or less in width or height?

Even though sometimes the data can't be readily seen in a single screen in Eclipse, we will generally consider debugger to be able to display all input/output information about one method call on a single screen. Screen resolution on the system tested was 1366x768.

As we can very quickly change zoom and pan in VERDICTS, we have adjusted the zoom to satisfy PXL ("pixel size sufficient") in each test, and measured the corresponding density. As Eclipse doesn't use visualizations it also always satisfies this requirement, so we do not report this metric, which would always be "yes" throughout the table. Table 4.4 shows our results. Most significant difference is observable in control flow density. For observing control flow, VERDICTS, on average, revealed 236 times more method calls per page viewed!

[5]If hovering by mouse reveals more information, we will only count what can be gleaned in 4-5 seconds.

Table 4.4: VERDICTS integrates large amounts of program values in comprehensive but useful ways

VERDICTS Tests	ST	CTRL	DATA	Eclipse Tests	ST	CTRL	DATA
JUnit1	4	106	1		0	1	1
JUnit2	1	112	3		1	1	1
Ant1	2	167	1		0	1	1
Ant2	1	451	2		0	1	1
Commons1	3	200	1		2	1	1
Commons2	8	380	15		2	1	1
VERDICTS1	2	142	1		0	1	1
VERDICTS2	1	330	4.5		0	1	1
Average	2.8	236	3.6		0.6	1	1

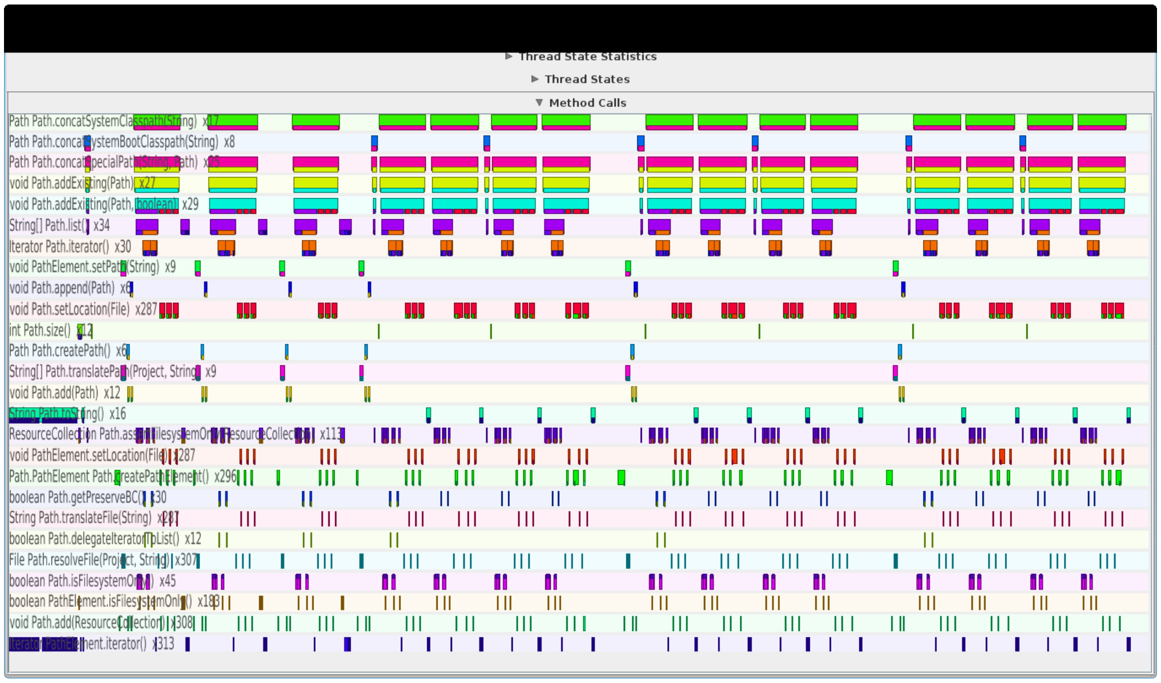

▸ Thread State Statistics
▸ Thread States
▾ Method Calls
Path Path.concatSystemClasspath(String) x17
Path Path.concatSystemBootClasspath(String) x8
Path Path.concatSpecialPath(String, Path) x25
void Path.addExisting(Path) x27
void Path.addExisting(Path, boolean) x29
String[] Path.list() x34
Iterator Path.iterator() x30
void PathElement.setPath(String) x9
void Path.append(Path) x6
void Path.setLocation(File) x287
int Path.size() x12
Path Path.createPath() x6
String[] Path.translatePath(Project, String) x9
void Path.add(Path) x12
String Path.toString() x16
ResourceCollection Path.assertFilesystemOnly(ResourceCollection) x113
void PathElement.setLocation(File) x287
Path.PathElement Path.createPathElement() x296
boolean Path.getPreserveBC() x30
String Path.translateFile(String) x287
boolean Path.delegateIteratorToList() x12
File Path.resolveFile(Project, String) x307
boolean Path.isFilesystemOnly() x45
boolean PathElement.isFilesystemOnly() x183
void Path.add(ResourceCollection) x308
Iterator PathElement.iterator() x313

Figure 4.19: VERDICTS test for Ant2 had an average of about 451 method calls visible clearly per screen

Fig.4.19 shows a VERDICTS screenshot from Ant2 test, which had an average of 451 method calls visible per screen without any method call shown below single pixel width or height. This is the highest value we have observed in our experiments.

For data density, we observed a large variation. The highest value was in Commons2, because these unit tests often use very simple short argument values such as:

```
coll.add("A");  coll.add("AA");
coll.add("B");  coll.add("BB");  coll.add("BA"); ...
```

Lower values in data density were observed when methods didn't have primitive or string arguments, or when string arguments were too long to view easily (such as full path to library JAR file). When objects passed as arguments had a standard string representation (Java's toString() method), if the string representation was useful and not too long, the data density could be higher than 1 call per screen.

4.9.4 B. Using Novel Visualizations That Reveal Patterns In Control Flow And Data Variations

For this feature, we measured time and output. We used three original metrics, and two derived metrics:

- T: Time (mins): Time spent to analyze and discover control flow patterns and variations in data (in minutes; 40 minutes max).
- PAT: Patterns: Number of control flow and data variation patterns observed in this time.
- CALL: Method Calls: Number of method calls examined/observed/analyzed in this time.
- PAT/min: Patterns per minute = PAT/T
- CALL/min: Method calls per minute = CALL/T

Original raw metrics for these tests are shown in table 4.5, and derived metrics are shown in table 4.6.

Table 4.5: Raw metrics from tests to prove VERDICTS reveals patterns in control flow and data variations.

VERDICTS Tests	T	PAT	CALL	Eclipse Tests	T	PAT	CALL
JUnit1	6.2	12	106		7.8	9	73
JUnit2	8.2	10	56		38.9	5	27
Ant1	3.9	13	167		12.5	10	74
Ant2	17.2	20	2708		20.8	17	112
Commons1	7.6	8	40		16.8	6	55
Commons2	9.8	10	1887		20.6	4	62
VERDICTS1	9.3	20	2704		40.1	18	87
VERDICTS2	13.1	13	404		40.2	21	188
Average	9.4	13.3	1054		24.7	11.3	84.8

Table 4.6: VERDICTS uses novel visualizations that reveal patterns in control flow and data variations.

VERDICTS Tests	PAT /min	CALL /min	Eclipse Tests	PAT /min	CALL /min
JUnit1	1.9	17.1		1.2	9.4
JUnit2	1.2	6.8		0.1	0.7
Ant1	3.3	42.8		0.8	5.9
Ant2	1.2	157.4		0.8	5.4
Commons1	1.1	52.8		0.4	3.3
Commons2	1.0	192.6		0.2	3.0
VERDICTS1	2.2	290.8		0.4	2.2
VERDICTS2	1.0	30.8		0.5	4.7
Average	1.6	98.9		0.6	4.3

Of most interest are the derived metrics shown in 4.6, patterns per minute and calls per minute. VERDICTS averaged 1.6 patterns observed per minute, and these patterns were observed over an average of 98.9 calls per minute. Eclipse averaged 0.6 patterns observed per minute, and these patterns were only observed over an average of 4.3 calls per minute. In terms of calls observed per minute, VERDICTS was, on average, 23 times faster than Eclipse.

As Eclipse traversal was too slow at times, we had to remove some breakpoints at times after seeing a general pattern. But this caused some patterns to be partial patterns as some of the calls were missing in the pattern. We counted partial patterns as half a pattern.

The patterns that are readily visible in VERDICTS are not readily visible in Eclipse as we have to go through depth-first traversal of method calls in Eclipse when trying to observe all patterns between all method calls. Even though recording the patterns is not included in this time, Eclipse without recording the partial patterns does not really allow one to see the patterns because of many other lower level calls seen mid-pattern.

4.9.5 C. Dynamically Inserting Probes And Hypotheses About Program Behavior Using A Familiar Language

For this feature, we used four metrics:

- TV: Time to verify: Time taken to verify one hypothesis/observation (in VERDICTS: one contract assertion).
- CLK: Clicks: Number of mouse clicks per hypothesis
- L: Lines: Number of lines of code written per hypothesis
- CALL: Method Calls: Number of method calls that this hypothesis was verified on

Our results are shown in table 4.7.

In these experiments, an effort was made to use similar complexity hypotheses, and sometimes equivalent hypotheses in VERDICTS and Eclipse. Different hypotheses can be arbitrarily different in complexity so lines should not be compared with each other. The values reported are quite often averages of two tests (two hypotheses verified).

In these tests, VERDICTS required, on average, 25% more lines of code. VERDICTS tests almost always took less time and needed fewer mouse clicks compared to Eclipse. The

Table 4.7: VERDICTS allows verifying hypotheses about program behavior using a familiar language

VERDICTS Tests	TV	CLK	L	CALL	Eclipse Tests	TV	CLK	L	CALL
JUnit1	8.9	58	3.5	5		5.4	82.5	1	5
JUnit2	2.4	52	0	31		1.3	23	0.5	31
Ant1	2.1	33	0.5	4.5		3.5	28	0	4.5
Ant2	6.1	35.7	3.7	34		13.4	161	4	21
Commons1	2.7	14.5	1	5		2.8	28.5	0	4
Commons2	1.8	21	1	286		5.9	53	1.5	23
VERDICTS1	2.1	7	0.3	559		3.6	41	1	9
VERDICTS2	6.7	33.5	3.5	21		9.3	36.5	2.5	21
Average	4.1	31.8	1.7	118.2		5.7	56.7	1.3	14.8

most significant difference appeared again when there were more method calls to observe. VERDICTS could on average verify the hypothesis in eight times more calls, in shorter time.

In these three subsections, we have seen how VERDICTS performs significantly better than Eclipse on three important features of VERDICTS:

- A. Integrating large amounts of program values in comprehensive but useful ways
- B. Using novel visualizations that reveal patterns in control flow and data variations
- C. Dynamically inserting probes and hypotheses about program behavior using a familiar language

VERDICTS provides automation through contracts, high-level view of the software through its visualizations, and numerical and statistical aggregate analysis and viewing through boxplots, correlation matrix, and X-Y plots. Our tests show that when analyzing a large number of method calls, there is a very pronounced difference between manual drudgery of using a modern IDE and using VERDICTS.

VERDICTS was able to show, on average, 236 times more method calls per screen for control flow analysis, and data from about 3.5 times more method calls per screen for data analysis. VERDICTS was on average 23 times faster in helping us analyze method calls, and helped us observe almost three times as many dependable control flow and data patterns. For hypothesis verification, VERDICTS required significantly fewer mouse clicks, and verified our hypotheses against eight times more method calls while taking 28% less time than using Eclipse.

The advantages of VERDICTS are most pronounced when the program analyzed or tested has a very high number of method calls. Our experiments show that VERDICTS is a very efficient tool to help a developer make sense of and verify dynamic behavior of large software.

Chapter 5

SMT: Semantic Mutation Testing

5.1 Specification-Implementation Concordance and Measuring Verifier Adequacy

5.1.1 Importance of Up-To-Date Specifications

Software development is a structured creative process of human-machine communication and control. The contract between the end user and the software developers is the requirements specification document, without which, the end users would have to manually verify every aspect of the implementation. As we stated before, documented specifications are central to software verification and validation, and are of great importance to the tasks of software comprehension, maintenance and reuse. Software and component functional requirements often constitutes the main bulk of any requirements specification document. In our research we also focus on functional requirements, software and component inputs and outputs.

Often, software performs as expected but the documented specifications have fallen out of sync with respect to evolving implementation. Both discovery of missing specifications and corrective maintenance of incorrect specifications require measuring the quality of specifications, not in terms of accurately representing end user needs, but rather, to answer this question of concordance:

> *"How faithfully (accurately) does this specification represent the behavior of this implementation?"*

Such a measure can help us discover incompleteness in existing specifications with respect to actively maintained evolving implementation. If implementation exists and works as expected but the specifications are missing and need to be discovered, alternative versions of (hypotheses for) specifications can be compared by this same measure.

Specification-implementation concordance measures adequacy of specification against the implementation, and can only represent actual end-user need to the degree that the program is already known to perform correctly as judged by the end user. For component reuse, this is exactly the quality that we need to measure and improve.

Can we measure specification-implementation concordance just by using these specifications to verify the software as it is implementated? Although this approach can quickly reveal false negatives, it cannot reveal false positives:

- False negative (incorrect fail): Software is correctly implemented, but specifications are too strict, and fail software.

- False positive (incorrect pass): Software is incorrectly implemented, but specifications are too lax and pass software.

Even if program under test is validated by users, and the program passes all tests, this still says nothing about the strictness of functional specifications and the adequacy of the corresponding oracle. False oracles generated from weak functional specifications can generate false positives and instill a false sense of trust in the program.

This point is easy to observe when we consider the extreme case of maximally incomplete (maximally lax) empty specification that passes (verifies as correct) any implementation. Passing such a specification obviously does not mean the software as implementated is adequate for any purpose, and therefore, this specification is inadequate in verifying any program as well. At the ideal extreme, a strict verifier would only accept correct outputs from the program, and would not accept even minor faults that cause small perturbations in the target program's outputs.

Incompleteness due to missing specifications can be a problem even when specifications are recorded as automatic verifiers for the software, and these verifiers continue to validate the implementation. Researchers report that incompleteness of specifications can create subtle change in behavior [12] [1] and can be hard to discover [60].

5.1.2 Human-Language Specifications and Automated Verifiers

Use of human languages in requirements specifications documents have caused specification quality research to put more emphasis on human-centric criteria, mainly for comprehension and implementation of specifications [12] Traditionally, human testers interpret this document to verify the program through manual tests or automated tests. The interpretation stage often reveals shortcomings of human language, such as ambiguity and inconsistency.

Today's full suites of automated tests replace much manual testing, allow agility in software development (developer can change code without fear of undetected breakage), and enable the continuous integration paradigm.

It is practically impossible to evaluate and measure adequacy of specifications written in an ambiguous language. We can eliminate ambiguity if we can convert such specifications to automated verifiers (passive Oracles). Sometimes, automated verifiers are in fact the primary form in which the specifications are recorded and maintained. If formal specifications are used instead, they can be used to generate automated verifiers as well.

Fig.5.1 shows how an automated verifier checks a program. Instead of memorized test input-output tuples that constitute a test suite, a verifier depends on a test input generator. In this case, the verifier, together with a set of inputs, is equivalent to a test suite in purpose and function, except for the possibility that a verifier may accept multiple outputs for the

[1] Black, et al [12] show this indirectly, by noting that missing boolean condition mutants of specifications are not easily discovered to behave differently from original program by most tests.

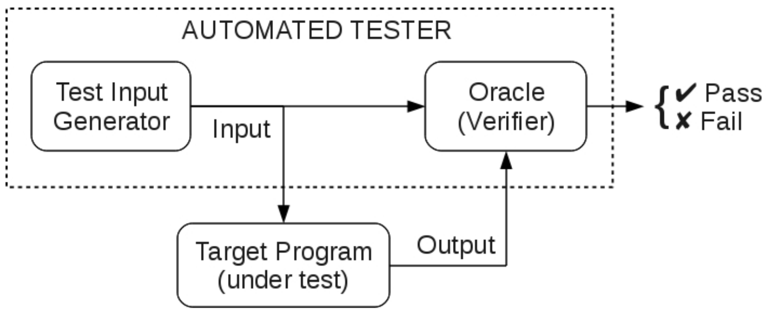


Figure 5.1: Automated tester, with decoupled verifier

same input (and therefore may allow nondeterministic behavior).

Compared to a test suite, a verifier is more generally applicable as it can work with any input. With a verifier, we can change the number of test cases quickly by using random or feedback-directed random [67] input set generation. The verifier can also be left in the program for in-field verification of the program or component.

5.1.3 Design by Contract

As we have seen in section 3.3.1, Design by Contract (DBC) is a method of specification and automated (inlined, executable) verification of object-oriented components. DBC is Bertrand Meyer's answer to the questions and issues of ambiguity and inconsistency that are prevalent in human language specifications [60].

Meyer's DBC specifications (called "contracts") add executable side-effect-free checks before each method to require valid inputs ("preconditions") and after, to ensure correct functional operation ("postconditions"). Every method must also preserve, upon termination, the object internal consistency conditions ("object invariants").

Preconditions and postconditions are popular constructions outside DBC as well. In UML modeling of object-oriented classes, method behavior can be specified by using object constraint language (OCL), which also depends on pre- and post-conditions. Z language for formal specifications also declares preconditions and postconditions to abstractly define operations.

Consider this div method, with a human-readable but not automatically testable API documentation:

```
/** Divides n by m; m should not be 0. */
int div(int n, int m) { return n/m; }
```

A partial contract for div may only allow operation with nonnegative n and positive m values. This is a contract for a partial domain (input value set). We mark preconditions with @pre and postconditions with @post:

```
@pre   n >= 0  &&  m > 0
@post  $result * m >= n
@post  ($result - 1) * m < n
```

A more general complete contract can easily be derived by requiring these conditions on absolute values of n and m (leaving only m != 0 as the precondition), as well as requiring consistency of sign in $result. Different syntaxes and languages can be used for DBC; for example, UML diagrams use OCL (Object Constraint Language) for such specifications. We use DBC in our experiments, but this can easily be switched with any other form of automatically verifiable specification.

5.1.4 Test Suite Adequacy

Roman poet Juvenal's latin phrase "Quis custodiet ipsos custodes?", best translated as "who watches the watchmen?", expresses that it may be naive to assume that judges and enforcers themselves are good.

In software testing, this corresponds to the question "who tests the tester?" For automated software testing with test suites, the general question becomes "who tests the test suite?". This is a question of test suite adequacy, and various test suite adequacy criteria are proposed to answer this question.

Test suite adequacy criteria do not depend on an outside third-party arbiter. Instead, all test suite adequacy criteria answer this question with “the program (its specific implementation) tests the test suite”. An important distinction is that test suite adequacy criteria depend on the program’s instructions, not only its black-box behavior. Differently implemented equivalent-behavior programs may produce different test suite adequacy scores for the same test suite. Merely using the compiler for the programming language of implementation is not sufficient; we need to use a separate source code analysis tool specific to the test suite adequacy criterion used.

5.1.5 Specification Adequacy

Specification-implementation concordance is a measure of specification adequacy. Can we use test suite adequacy criteria to measure specification-implementation concordance as well? Two popular approaches for test suite adequacy are:

- Code coverage criteria (various criteria)
- Mutation Testing (Mutation Analysis)

Code coverage criteria (such as statement coverage, branch coverage, and multiple condition coverage) only reveal input variability and how inputs exercise various statements, decisions, branches and blocks of the program, but they completely ignore program’s outputs and verifier’s pass/fail verdict. In fact, code coverage criteria does not fully evaluate any test suite either; they only evaluate the set of test inputs, while completely ignoring the corresponding expected outputs.

In fig.5.1, code coverage criteria would evaluate the test input generator with respect to the target program while completely ignoring both the program’s output and the verifier. Code coverage criteria scores would not change at all if we replaced the verifier with an empty verifier that passes any implementation. Code coverage criteria therefore cannot be used to expose false oracles, weak functional specifications and verifiers.

Mutation Testing [65] (MT, "traditional MT") subsumes many types of code coverage criteria [66] MT also depends on the pass/fail judgment of the verifier, so it may potentially be used to measure the adequacy of verifiers and the functional specifications they represent.

Various researchers report a number of serious shortcomings of MT. In section 5.4, we will examine some of their findings, as well demonstrate some of the problems with a small example.

5.1.6 Measuring Specification Adequacy with Semantic Mutation Testing

Mutation testing traditionally mutates keywords and symbols of the source code, often without considering how dramatically such a mutation may change behavior. We consider traditional mutation testing "syntactic" mutation testing.

In [14] , we introduced Semantic Mutation Testing (SMT) to measure component specification-implementation concordance. SMT borrows ideas from traditional MT but controls and limits the perturbation of variable values that collectively hold the program dynamic state, so that mutations generate a small variation in the overall program behavior. In black-box SMT, error is injected only to the inputs and outputs of the program. In white-box SMT, error can be injected in any expression that evaluates a value that can be mutated so long as such a mutation cannot cause the program to crash.

As we do not introduce human validation, the adequacy we can measure represents verifier-implementation concordance. When automated verifiers are used as the primary form of specification, or when primary specification is correctly converted to a verifier, SMT adequacy score also measures the adequacy of the specifications.

In section 5.2, we define a subsumption relation (a partial order) for functional specifications and verifiers. In section 5.3, we use this subsumtion relation to express desired requirements for a proper specification/verifier adequacy score. Section 5.4 introduces traditional MT and examines its shortcomings. In section 5.5, we introduce our Semantic MT approach, explain the black-box and white-box SMT, and discuss why and how we proactively use data flow analysis to avoid crashes due to corrupted internal state.

5.2 Functional Specifications and Verifiers: A Subsumption Relation

5.2.1 Functional Specification and Verifier Adequacy

Consider program $P : \mathbb{I} \rightarrow \mathbb{O}$, defined as a function from the set of all possible valid inputs $\mathbb{I}$, to the set of all possible valid outputs $\mathbb{O}$. For the same set of inputs and outputs, there can be multiple programs (implementations). Multiple such programs may have the correct behavior with respect to a given set of requirements.

A verifier for functional requirement specifications, as seen in fig.5.1, confirms that for given inputs, program's outputs represent expected behavior.

Mathematically, a verifier is a predicate, a Boolean-valued function from the set $\mathbb{I} \times \mathbb{O}$ to {true, false}, where true represents verifying that the outputs conform to functional requirements for the given inputs. Everything we define below for verifiers also holds for the functional specifications that the verifiers represent.

For given input and output sets $\mathbb{I}$ and $\mathbb{O}$, we will denote the set of all possible verifiers with $\mathbb{V}_{\mathbb{I}\times\mathbb{O}}$. In the following, we are only concerned with comparing verifiers from the same set $\mathbb{V}_{\mathbb{I}\times\mathbb{O}}$.

As an example, consider a function that returns the double-valued average of values (output: avg) for an integer array (input: ar), implemented in a programming language L. $\mathbb{I}$ is the set of all possible integer arrays for L (depends on the integer type for L), $\mathbb{O}$ is all possible double values that averages of such integer arrays could produce. For input array ar and returned value avg, we can have alternative verifiers V_1 and V_2 which implement and imply alternative functional specifications:

- V_1: avg is the average of values in ar
- V_2: avg is no less than the minimum value in ar

In this case, V_1 and V_2 are comparable as average value cannot be less than the minimum value for a data set. V_1 is a more strict verifier compared to V_2. In the following, we will define a strict ordering where we formalize this as "strict subsumption" relation. In this example, we observe that "V_1 strictly subsumes V_2" (and the functional specification implemented by V_1 strictly subsumes the functional specification implemented by V_2). Mathematically we will express this as $V_1 > V_2$.

The concept of a subsumption relation is also used to compare test adequacy criteria [66] and both subsumption and strict subsumption relations were defined in [44] to compare mutants of a program.

5.2.2 A Formal Definition for Subsumption Relation Between Verifiers

Definition: Where P: $\mathbb{I} \to \mathbb{O}$ is a program and V: $\mathbb{I} \times \mathbb{O} \to$ {true, false} is a verifier, we define "V accepts/passes P" as:

$$pass(V, P) := \forall x \in \mathbb{I} : V(x, P(x)) = true$$

If the verifier V does not pass program P, we will express this with fail(V, P):

$$fail(V, P) := \exists x \in \mathbb{I} : V(x, P(x)) = false$$

Definition: Verifier V_1 *is subsumed by* verifier V_2 (expressed as $V_1 \leq V_2$) iff $\forall x \in \mathbb{I}, y \in \mathbb{O} : V_2(x, y) \implies V_1(x, y)$. Equivalently, we can also use $V_2 \geq V_1$, which can be read V_2 *subsumes* V_1.

Even though we will conform to the traditional use of the symbol $\leq$ to define an "is subsumed by" partial order (and later, a lattice) on verifiers, the dual, $\geq$ is conceptually simpler and easier to read as seen in the definition above.

For $V_2 \geq V_1$, we have:

$$(\forall x \in \mathbb{I}, V_2(x, P(x)) = true) \Longrightarrow (\forall x \in \mathbb{I}, V_1(x, P(x)) = true)$$

Therefore:

$$pass(V_2, P) \wedge V_2 \geq V_1 \Longrightarrow pass(V_1, P)$$

V_2 is more strict than, or at least as strict as V_1, and the $\geq$ symbol represents that V_2 is a better (or at least equivalent) verifier for a program P as long as we have $pass(V_2, P)$.

Definition: Verifier $V_1 = V_2$ if $V_1 \leq V_2$ and $V_2 \leq V_1$.

Note that this definition of equivalence depends on evaluation of V_1 and V_2 for all possible inputs and outputs. There may be different but equivalent expressions of specifications and different but equivalent implementations of verifiers. We are not concerned with symbolic construction, syntax and internal structure of specifications and verifiers; we are only concerned with the semantics, evaluation and behavior.

5.2.3 Strict Subsumption of Verifiers

Definition: Verifier V_1 *is strictly subsumed by* verifier V_2 (expressed as $V_1 < V_2$) iff $V_1 \leq V_2 \wedge V_1 \neq V_2$. Equivalently, we can express this with $V_2 > V_1$ and state that V_2 *strictly subsumes* V_1.

As $V_1 \leq V_2$, the only way to have $V_1 \neq V_2$ is to have $V_2 \nleq V_1$. This implies that:

$$\exists z_0 \in \mathbb{I} \times \mathbb{O} : \neg(V_1(z_0) \Longrightarrow V_2(z_0))$$

$$\exists z_0 \in \mathbb{I} \times \mathbb{O} : V_1(z_0) \wedge \neg V_2(z_0)$$

In terms of the program's inputs and outputs, this means that there is at least an input x_0, and an output y_0 for which $V_1(x_0, y_0)$ is true but $V_2(x_0, y_0)$ is false.

Consider the case $V_1 < V_2$ for a program P with pass(V_1, P). There must be some input x_0, and output y_0 for which $V_1(x_0, y_0)$ is true and $V_2(x_0, y_0)$ is false. We can create a program P' that corresponds to:

```
P'(input):
  if (input == x0)
  then return y0;
  else return P(input);
```

This program only differs from P in returning y_0 for x_0. We will now have pass(V_1, P'), but fail(V_2, P').

Can there be another P" that causes the reverse; fail(V_1, P") and pass(V_2, P")? As $V_1 \leq V_2$, pass(V_2, P") $\implies$ pass(V_1, P") for any program P", so there is no example to the contrary. Therefore $V_1 < V_2$ implies that V_1 is strictly less discriminating than V_2, as it cannot detect that a program P' behaves differently from P whereas V_2 can.

5.2.4 Least Upper Bound (Supremum) And Greatest Lower Bound (Infimum)

Next, we would like to show that in the set of verifiers $\mathbb{V}_{\mathbb{I}\times\mathbb{O}}$, any two verifiers have a least upper bound (supremum) and a greatest lower bound (infimum).

We will first define our join and meet operators, $\wedge$ and $\vee$ [2], then show that these operators actually give us the supremum and infimum for our partial order. In mathematical terms, this would allow us to declare that our partial order is a lattice.

[2]Traditionally, partial order, join and meet symbols are ($\leq$, $\vee$, $\wedge$). We instead use ($\leq$, $\wedge$, $\vee$) here, and the dual lattice with reverse partial order is ($\geq$, $\vee$, $\wedge$). Use of reverse symbols stems from our desire to place more strict verifiers higher in the partial order and the lattice.

Definition: Given V_1, $V_2 \in \mathbb{V}_{\mathbb{I}\times\mathbb{O}}$, we define join operator for verifiers, $\wedge$, as universally quantified logical and:

$$V = V_1 \wedge V_2 \iff \forall z \in \mathbb{I} \times \mathbb{O} : V(z) = V_1(z) \wedge V_2(z)$$

To prove that our join operator indeed conforms to what is expected of a join operator for the partial order $\leq$, we need to prove that, for $V_3 = V_1 \wedge V_2$, we have:

- $V_1 \leq V_3$
- $V_2 \leq V_3$
- $\forall V \in \mathbb{V}_{\mathbb{I}\times\mathbb{O}} : (V_1 \leq V) \wedge (V_2 \leq V) \implies V_3 \leq V$

From $P \wedge Q \implies P$, we have:

$$\forall z \in \mathbb{I} \times \mathbb{O} : V_1(z) \wedge V_2(z) \implies V_1(z)$$

$$\forall z \in \mathbb{I} \times \mathbb{O} : V_3(z) \implies V_1(z)$$

$$V_1 \leq V_3$$

As $\wedge$ is commutative, the same holds for V_2:

$$V_2 \leq V_3$$

For the last requirement of being a join (that this is actually the *least* upper bound), note that, in general, we have:

$$(P \implies Q) \wedge (P \implies R) \implies (P \implies (Q \wedge R))$$

Specifically, for any V $\in \mathbb{V}_{\mathbb{I}\times\mathbb{O}}$, we have:

$$\forall z \in \mathbb{I} \times \mathbb{O} : (V(z) \implies V_1(z)) \wedge (V(z) \implies V_2(z)) \implies (V(z) \implies (V_1(z) \wedge V_2(z))$$

$$\forall z \in \mathbb{I} \times \mathbb{O} : (V(z) \implies V_1(z)) \wedge (V(z) \implies V_2(z)) \implies (V(z) \implies V_3(z))$$

$$(V_1 \leq V) \wedge (V_2 \leq V) \implies (V_3 \leq V)$$

Definition: Given V_1, $V_2 \in \mathbb{V}_{\mathbb{I}\times\mathbb{O}}$, we define meet operator for verifiers, $\vee$, as universally quantified logical or:

$$V = V_1 \vee V_2 \iff \forall z \in \mathbb{I} \times \mathbb{O} : V(z) = V_1(z) \vee V_2(z)$$

To prove that our meet operator indeed conforms to what is expected of a meet operator for the partial order $\leq$, we need to prove that, for $V_4 = V_1 \vee V_2$, we have:

- $V_4 \leq V_1$
- $V_4 \leq V_2$
- $\forall V \in \mathbb{V}_{\mathbb{I}\times\mathbb{O}} : (V \leq V_1) \wedge (V \leq V_2) \implies V \leq V_4$

From $P \implies (P \vee Q)$, we have:

$$\forall z \in \mathbb{I} \times \mathbb{O} : V_1(z) \implies (V_1(z) \vee V_2(z))$$

$$\forall z \in \mathbb{I} \times \mathbb{O} : V_1(z) \implies V_4(z)$$

$$V_4 \leq V_1$$

As $\vee$ is commutative, the same holds for V_2:

$$V_4 \leq V_2$$

The third requirement for meet is that it indeed gives us the *greatest* lower bound. Observe that logical operator $\vee$ distributes over $\wedge$:

$$(P \vee R) \wedge (Q \vee R) \equiv (P \wedge Q) \vee R$$

Using P' = ¬ P and Q' = ¬ Q, and using $P \implies Q \equiv \neg P \vee Q$ we have:

$$(P' \implies R) \wedge (Q' \implies R) \equiv (P' \vee Q') \implies R$$

Specifically, $\forall V \in \mathbb{V}_{\mathbb{I}\times\mathbb{O}}$, we have:

$$\forall z \in \mathbb{I} \times \mathbb{O} : (V_1(z) \implies V(z)) \wedge (V_2(z) \implies V(z)) \implies ((V_1(z) \vee V_2(z)) \implies V(z))$$

$$\forall z \in \mathbb{I} \times \mathbb{O} : (V_1(z) \implies V(z)) \wedge (V_2(z) \implies V(z)) \implies (V_4(z) \implies V(z))$$

$$(V \leq V_1) \wedge (V \leq V_2) \implies V \leq V_4$$

Therefore:

- Partial order $\leq$, defined over verifiers is a lattice with the join operator $\wedge$ and the meet operator $\vee$ defined above
- Any two verifiers V_1 and V_2 have:
  - A supremum (least upper bound) $V_3 = V_1 \wedge V_2$
  - An infimum (greatest lower bound) $V_4 = V_1 \vee V_2$

5.2.5 Bottom Element, Top Element, and Complete Verifier for a Program P

Next, we will define and examine bottom and top elements in our set of verifiers $\mathbb{V}_{\mathbb{I}\times\mathbb{O}}$, with respect to partial order $\leq$; these are the least and most strict verifiers compared to all other verifiers.

Definition: Let us define two verifiers $\top_{\mathbb{I}\times\mathbb{O}}$ that is always false, and $\bot_{\mathbb{I}\times\mathbb{O}}$ that is always true, for any program input and output:

$$\forall z \in \mathbb{I} \times \mathbb{O} : \top_{\mathbb{I}\times\mathbb{O}}(z) = false, \bot_{\mathbb{I}\times\mathbb{O}}(z) = true$$

For any proposition P, we have $false \implies P$ and $P \implies true$. By universally quantifying these for any z in $\mathbb{I} \times \mathbb{O}$, we can easily show that these two verifiers, $\top_{\mathbb{I}\times\mathbb{O}}$ and $\bot_{\mathbb{I}\times\mathbb{O}}$, are the top and bottom elements of the lattice $(\mathbb{V}_{\mathbb{I}\times\mathbb{O}}, \leq)$ with partial order $\leq$, and the join and meet operations defined earlier:

$$\forall V \in \mathbb{V}_{\mathbb{I}\times\mathbb{O}} : \bot_{\mathbb{I}\times\mathbb{O}} \leq V \wedge V \leq \top_{\mathbb{I}\times\mathbb{O}}$$

The top and bottom element conceptually correspond to most strict and most lax verifiers and specifications. Specifically:

- $\top_{\mathbb{I}\times\mathbb{O}}$ is always false. For any program P, we have fail($\top_{\mathbb{I}\times\mathbb{O}}$, P). Program P is not allowed to do anything.
- $\bot_{\mathbb{I}\times\mathbb{O}}$ is always true. For any program P, we have pass($\bot_{\mathbb{I}\times\mathbb{O}}$, P). Program P is allowed to do anything; it is not required to do anything specific.

Neither top nor bottom element is useful for verifying an actual program. For any program P, the least strict valid verifier (one that passes P) is the bottom element. But top is not the counterpart; it is not the most strict valid verifier as it does not pass any program P. What then, is the most strict verifier for a given program?

Definition: For a given program P, consider the set of all verifiers that pass P. We define V_P, the *complete verifier* for P as the supremum (least upper bound, join) of all verifiers that pass P. Then, any verifier V that passes P must have $V \leq V_P$:

$$\forall V \in \mathbb{V}_{\mathbb{I} \times \mathbb{O}} : pass(V, P) \Longrightarrow V \leq V_P$$

5.3 Measuring Verifier Adequacy

5.3.1 Can Verifier Distinguish Between Correct and Faulty Implementations?

A verifier for a program can often be written in the same programming language as the program. For example, consider a function fn that takes one object of type Cls, and returns an integer value, written in a C-like object-oriented language such as Java or C++:

```
int fn(Cls obj) {
  int n = obj.length();
  return n * n;
}
```

A verifier for such a function would take both the inputs and the return value to give a pass/fail judgement:

```
boolean verify_fn(Cls obj, int ret) { ... }
```

Consider these three variants of fn that are improperly implemented:

```
int fn1(Cls obj) { int n = obj.length(); return 0;     }
int fn2(Cls obj) { int n = obj.length(); return n;     }
int fn3(Cls obj) { int n = obj.length(); return n + 3; }
```

A good verifier should pass fn, but fail fn1, fn2 and fn3.

This example shows clearly what we stated before: Code coverage criteria cannot be used to compare verifiers. The source code of improper implementations is the same as properly implemented function except for the returned value. In any function, even when the same exact decisions are made, same branches are taken and code blocks and same set of instructions are evaluated, the data, and the value returned can be different.

Traditional MT can be used to evaluate verify_fn by creating mutants of fn that verify_fn should not pass. We will show in section 5.4.7 that MT may create mutants that differ wildly from the original program. This makes it hard to discover whether our verifier can detect subtle changes in program behavior. In section 5.5, we will explain our technique for creating subtly mutated test runs for fn. In our tests, we found two variants of SMT to be better than MT in comparatively evaluating adequacy of competing verifiers for a program.

To be able to compare MT and SMT, we need to first examine the more general question: How can we evaluate quality of approaches to create a verifier adequacy score? Next section tries to create a very basic set of baseline requirements for verifier adequacy score.

5.3.2 Verifier Adequacy Score: Requirements

Verifier adequacy score depends on the program P_0 that is being verified. With respect to the $\leq$ ordering of verifiers, the bottom element $\bot_{\mathbb{I}\times\mathbb{O}}$ is always true for any program input and output, checks nothing, and accepts any program. Compared to all other verifiers that pass the program, bottom element is the least discriminating verifier for any program P_0, and should get a score 0.0.

As we saw before, no program has as its best verifier the top element $\top_{\mathbb{I}\times\mathbb{O}}$, which fails any program. As an over-zealous verifier that fails the program, $\top_{\mathbb{I}\times\mathbb{O}}$ should probably also get a score of 0.0. This is a fail fast scheme, that allows quick realization when verifier is evolved to be too strict for the program at hand.

The most strict verifier that passes P_0 is the complete verifier defined earlier, V_{P_0}, which is the supremum of all verifiers that pass P_0. V_{P_0} should get adequacy score 1.0.

Now we can state what we expect from a good verifier adequacy score with respect to a given program P_0 (score(P_0, V)). We can only compare verifiers that are compatible with this program, so if P_0 is defined from $\mathbb{I}$ to $\mathbb{O}$, then we can compare scores for verifiers defined in $\mathbb{V}_{\mathbb{I}\times\mathbb{O}}$. The core requirements are:

1. Adequacy score should fully use range 0.0 - 1.0, so the bottom element (the empty verifier) $\perp_{\mathbb{I}\times\mathbb{O}}$ should get score 0.0: score(P_0, $\perp_{\mathbb{I}\times\mathbb{O}}$) $=$ 0.0

2. Most strict verifier that still passes P_0, the complete verifier V_{P_0} defined earlier, should get 1.0:

$$pass(V_{P_0}, P_0) \wedge \neg\exists V : (V > V_{P_0} \wedge pass(V, P_0)) \implies score(P_0, V_{P_0}) = 1.0$$

3. More strict verifiers that pass P_0 should get higher scores; strict subsumption implies strictly better score:

$$V_1 > V_2 \wedge pass(V_1, P_0) \implies score(P_0, V_1) > score(P_0, V_2)$$

4. Any verifier V that fails P_0 (and this includes the top element $\top_{\mathbb{I}\times\mathbb{O}}$) should get the score 0.0 (or fail with an error message):

$$fail(V, P_0) \implies score(P_0, V) = 0.0$$

The third requirement that strict subsumption implies strictly better score, together with first and second requirements also means that if a verifier that passes the program is not the bottom element (not always true), and not equivalent to complete verifier in behavior, it should get a score in the open range (0.0, 1.0):

- $V \neq V_{P_0} \wedge pass(V, P_0) \implies score(P_0, V) < 1.0$
- $V \neq \perp_{\mathbb{I}\times\mathbb{O}} \wedge pass(V, P_0) \implies score(P_0, V) > 0.0$

As is common with various test suite adequacy scores, verifier adequacy scores also are not comparable across metrics (or even, between programs with different $\mathbb{I} \times \mathbb{O}$), and a single score of 0.1 or 0.9 in one method of scoring does not mean anything. Any strictly monotonic mapping of [0,1] to [0,1] would preserve all these requirements. Specifically, distance of adequacy score from 1.0 need not correlate linearly with effort that remains in improving the verifier to get a score of 1.0.

Beyond strict subsumption, we may also like to require that a verifier that is harder to satisfy get a higher score, but this requirement is not easy to define as it depends on what we consider to be important. We can consider all of $\mathbb{I} \times \mathbb{O}$ equally important, consider use cases (or trace actual behavior) to create probability distribution for $\mathbb{I} \times \mathbb{O}$, or define input domains and consider values near domain boundaries as more important to verify by our verifier. Each approach gives different weight to different values in $\mathbb{I} \times \mathbb{O}$, and would produce different orders for verifiers. Although we attempt to order the verifiers in our experiments in perceived strictness, our order presumes all of $\mathbb{I} \times \mathbb{O}$ to be equally likely / equally important, which need not be implicitly assumed/followed by the methods we test (MT and SMT methods).

5.4 Traditional (Syntactic) Mutation Testing

5.4.1 MT Injects Small Syntactic Faults

We consider the traditional Mutation Testing (MT) as per [65] to be syntactic MT. More appropriately called "mutation analysis" (of test suites), this method does not test software, but rather provides a test criterion to evaluate test suite adequacy (quality), similar to various code coverage criteria.

For an original program, P with test cases T_1, T_2, T_3, ..., MT uses a predefined set of mutation operators (such as '/' $\rightarrow$ '+') to create "mutants" M_1, M_2, M_3, ... which are all possible single-mutation mutated versions of P. Mutants are then compiled and checked against the test suite. If mutant M_i does not pass the test suite (if any test case T_j fails), we consider mutant M_i killed. Otherwise mutant remains "live".

A mutant is semantically equivalent to P if it always behaves the same way as P for any input. No proper test (that passes correctly implemented program) could kill an equivalent mutant. Mutation adequacy score of a test suite is the number of mutants killed by the test suite divided by the number of mutants not equivalent to original program P. If this score is 1.0 (all non-equivalent mutants are killed) the test suite is called "mutation adequate".

5.4.2 MT Has High Computational Complexity

Due to the need for compilation of each mutant, MT has been prohibitively expensive. Offutt and Untch [65] state that suggested solutions to time complexity problem of MT falls under the general categories of "do fewer, do smarter, and do faster". So far, there is no universally agreed-upon solution to the time complexity problem of MT.

As a smarter approach to compiling mutants, Untch's "Mutant Schema Generation" approach [78] generates a single meta-mutant program that contains all mutants and is compiled only once. In white-box SMT (see section 4.2), we also do a single transformation to generate a meta-mutant containing all mutations. In our current implementation, we use bytecode manipulation and avoid compilation altogether.

5.4.3 A Major Hurdle: Semantic Equivalence of Mutants

As an example of semantic equivalence, these two loops behave the same way so long as i is not varied unpredictably from within the loop:

```
for(i=0; i  < 10; i++) ...
for(i=0; i != 10; i++) ...
```

Semantic equivalence of arbitrarily complex programs is undecidable; we cannot generally know if $M_i \equiv$ P. Proving that a mutant behaves the same exact way as the program under all circumstances is a tedious and error-prone manual task. Grün et al [36] report that for one target program they examined, 40% of the mutations turned out to be equivalent, and detecting equivalence for a single mutant took 15 minutes. In order to automate MT, Offutt and Untch suggest skipping mutant equivalence testing for some hard-to-analyze mutants [65]. But inadequacy of test suite is only discovered by mutants that pass all existing tests, and so, are suspected to be equivalent. If we skip equivalence testing, any test suite is "mutation adequate".

Most researchers quote seminal work by DeMillo in 1978 [28] as having started the field of MT. Even though programs mutated were orders of magnitude smaller, DeMillo, suggests but does not state that he did the manual equivalence checking of live mutants for the short example of Hoare's FIND function. The numbers show that DeMillo did not really check for semantic equivalence: He postulates at one point that the 14 live mutants for the final reduced test suite may be equivalent, but there cannot be more than 10 equivalent mutants because an earlier full test suite had only 10 live mutants left.

As equivalence is in general undecidable, and prohibitive in practice even for short programs, in traditional MT of nontrivial programs:

- Actual number of non-equivalent mutants is never truly known.
- Mutation adequacy score can only be approximated; it cannot be calculated.

5.4.4 Are Mutation Operators Competent?

There is a general acceptance in MT research that MT mutations represent potentially possible errors competent programmers may make. At the same time, MT researchers do not try to make mutation operators represent possible errors a competent developer may make, but rather try to use a large enough set of simple syntactic mutations to have a good battery of tests, ignoring the question whether some mutation operators may almost always produce dumb mutants.

For example, replacing one arithmetic operator with another or negating the whole condition in an if statement may change behavior for every possible input, whereas replacing < with <= or removing a subcondition within branching condition often keeps behavior unchanged for a large subset of inputs.

What is more, many mutation operators that are specifically designed to emulate and subsume various code coverage criteria have no place in normal programs. Consider, for example, a "fail" statement used merely to check statement coverage.

There is one way to measure whether a mutant generated by using one of the mutation operators can be considered relatively "competent" compared to the original program: If a mutant fails a significant number of test cases in our test suite, it differs too much in behavior compared to the original program, and therefore the mutation does not represent an error a competent programmer would miss, especially with today's availability of automated unit testing tools. But MT adequacy metric does not require measuring how much mutants differ in behavior compared to the original program.

If a mutant fails most of the test cases, its use in adequacy measure falsely inflates the metric. MT researchers never report (and in fact, never measure and never know) what proportion of mutants fail a majority of the test cases, so they do not know how competent a mutation operator or a mutant is.

As we will see in the next section, a rather large percentage of mutants (generated for programs with different sizes using common MT mutation operators) actually fail all tests, and are therefore maximally incompetent.

5.4.5 "All Mutants Are Equal" Myth and "Dumb Mutants"

Jia and Harman [44] highlight and argue against some serious faulty assumptions ("myths") of MT, as well as analyze and observe some problems with traditional MT in their higher order mutations (multiple syntactic mutations per mutant program) paper. These faulty assumptions are rarely stated, but often presumed correct in MT research.

Jia and Harman's "All Mutants Are Equal (AME) Myth" states that all mutated versions of a program are equally useful. This is easy to disprove: Many mutants fail 100% of the test cases. Jia and Harman call these "dumb mutants".

How prevalent are such dumb mutants? Jia and Harman tested MT for 10 programs of very different sizes, from 50 - 6,000 LOC, with 60 - 13,498 test cases. For these programs, 6% - 75% of all mutants generated were dumb mutants. Considering all mutants generated for all programs, 39,058 of 94,493 [3] mutants generated were dumb mutants; an average ratio of 41% of all mutants failed all tests.

For dumb mutants, any one test case (instead of 60 - 13,498 test cases used in Jia & Harman's test programs) is a mutation-adequate test suite all by itself! By appearing in both the numerator and the denominator, dumb mutants inflate the mutation adequacy score towards 1.0 and give a false sense of test suite adequacy.

Each mutant contributes equally to MT adequacy score, but many other researchers also suggest not every mutant is created equal.

Two recent papers [63] [77] use sampling from the population of all mutants (a "do fewer" approach) to both reduce the computational complexity and discover the most useful mutation operators with respect to affecting mutation adequacy score. [77] shows that good mutation operators introduce subtle or small changes, and for some values may evaluate to the same result (do not cause a state change for every input that reaches the mutation site). [63] shows that different mutation operators have very different level of correlation with the overall MT adequacy score, and clearly, should not be considered equivalent.

Clearly not every mutation operator and not every mutant should be considered equal. Best mutants do not always change program internal state, and change program state by a small amount when they do. Such mutants better highlight limitations of the unit tests

[3] Jia & Harman have a typo in their table 2. The last plot of fig 5, shows that the largest program they tested, "space", actually has 26,401 dumb mutants rather than 26,401 potentially equivalent mutants and 5,378 dumb mutants. Our totals are consistent with their fig 6, which shows totals across all programs. If we disqualify and skip "space", which has 13,498 test cases and 68,843 mutants, we instead get a rate of 49% dumb mutants.

(and the limitations of the verifier within the unit test). Our alternative method specifically focuses on keeping the disruptions to program state held in program variables to a minimum.

5.4.6 Semantic versus Syntactic Faults

Another myth defined by Jia and Harman is the "Syntactic Semantic Size (SSS) Myth," which states that programs generally have relatively few minor syntactic differences from a correct version. This is falsely derived from DeMillo's "Competent Programmer Hypothesis" [29] which states that a program written by a competent programmer will differ from a correct version by relatively few faults. But semantic differences from correct behavior (faults) are not necessarily caused by few syntactic differences; different algorithms, data structures and much refactoring may be needed for a small change in behavior.

Jia and Harman strongly argue that "few faults" (attributed to competent programmers and almost-correct programs) should refer to semantic operation/behavior similarity, and cannot be emulated by single-syntactic-change mutants. They show, through their experiments, that some multiple-syntactic-mutation mutants change behavior of the program more subtly than their constituent mutations do ("strict subsumption"). Such "higher order" mutants improve the quality of MT as compared to having each mutation in a separate mutant.

When MT was first proposed in 1978 [28] programs were significantly shorter, and were often written in a language that is intended to be at least partially readable by end users. The software development practices and computing machinery added significant delays between development and testing of a program. Because of these reasons, code reviews conducted by developers and end users were an important part of program verification.

Today, the computing machinery and the field have changed significantly, and faster machines now allow dynamic testing to often be a major component of testing. A large proportion of single-syntactic-difference mutants dramatically change program state for almost any input. Such mutants would not survive even a small amount of testing, and could not survive in competently written programs today.

5.4.7 Beyond "Dumb" Mutants: A Simple Example

SSS myth and “mutation operators are competent” assumption can be dispelled with a simple example [4]. Consider this method and its partial (incomplete) contract:

```
/*        int n0 = abs(n);
 * @post  $result <= 2*n0
 * @post  (abs(n) > 10 ||
 *         $result >= max(n0-2, 4*n0-20))
 */
int sqr5(int n) {
    if (abs(n) < 10) return n * n / 5;
    else return 20;
}
```

This is a somewhat strict specification that only allows values within [max(|n|-2, 4|n|-20), 2|n|] while $|n| \leq 10$. The actual outputs from this function compared to our contract's min and max requirements is shown in table 5.1.

Table 5.1: Input, output and contract limits for sqr5(int n)

| n |	0	1	2	3	4	5	6	7	8	9	10
sqr5(n)	0	0	0	1	3	5	7	9	12	16	20
max	0	2	4	6	8	10	12	14	16	18	20
min	-2	-1	0	1	2	3	4	8	12	16	20

Consider the very commonly used MT mutation operation of replacing an arithmetic operator in {*, /, %, +, -} with another. Fig.5.2 shows original function and eight mutants together with our requirements (contract). In fig.5.3, we see the values tabulated, and failure of contract highlighted in reverse video (black background). Of the eight mutants, only one

[4]Even though there are advantages to applying MT to large programs, it is much harder to examine and understand the utility and quality of mutants used by traditional MT in a large program. Larger programs have nonlinear behaviors and too many mutants to be examined closely.

(12.5%), n*n+5, fails all tests and is a dumb mutant. Even the two mutants that are always 0 do not fail all tests.

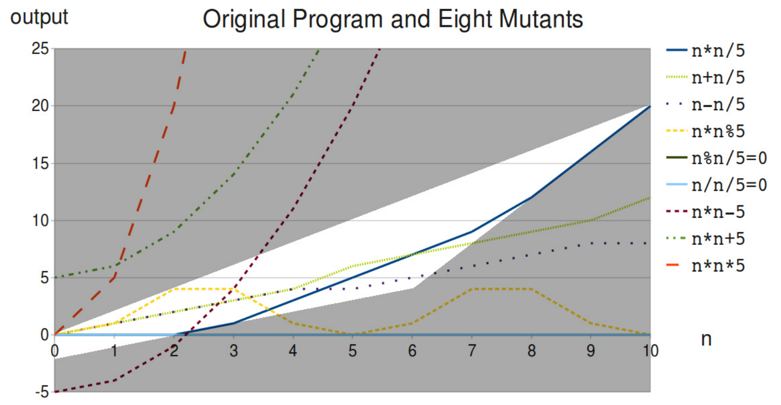
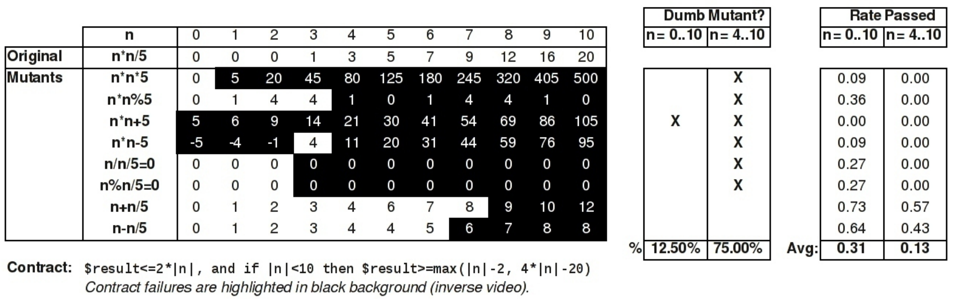


Figure 5.2: sqr5(int n) method (n*n/5) and eight mutants, compared to our somewhat strict requirements specifications (our contract, in white background)



	n	0	1	2	3	4	5	6	7	8	9	10	Dumb Mutant? n= 0..10	Dumb Mutant? n= 4..10	Rate Passed n= 0..10	Rate Passed n= 4..10
Original	n*n/5	0	0	0	1	3	5	7	9	12	16	20				
Mutants	n*n*5	0	5	20	45	80	125	180	245	320	405	500		X	0.09	0.00
	n*n%5	0	1	4	4	1	0	1	4	4	1	0		X	0.36	0.00
	n*n+5	5	6	9	14	21	30	41	54	69	86	105	X	X	0.00	0.00
	n*n-5	-5	-4	-1	4	11	20	31	44	59	76	95		X	0.09	0.00
	n/n/5=0	0	0	0	0	0	0	0	0	0	0	0		X	0.27	0.00
	n%n/5=0	0	0	0	0	0	0	0	0	0	0	0		X	0.27	0.00
	n+n/5	0	1	2	3	4	6	7	8	9	10	12			0.73	0.57
	n-n/5	0	1	2	3	4	4	5	6	7	8	8			0.64	0.43
												%	12.50%	75.00%	Avg: 0.31	0.13

Contract: `$result<=2*|n|, and if |n|<10 then $result>=max(|n|-2, 4*|n|-20)`
Contract failures are highlighted in black background (inverse video).

Figure 5.3: sqr5(int n) method (n*n/5) and eight mutants, showing contract failures in reverse video (black background). Dumb Mutants and the low rate of mutants passing tests is shown as well.

Mutants perform very poorly, and almost none of the mutants look like “a program written by a competent programmer that differs from a correct version by relatively few faults”. For n=10 where the correct output is 20, the output error for eight mutants are {-8, -12, -20, -20, -20, +75, +85, +480}.

A competently written program with few faults should pass most of the tests. Instead, for nontrivial test cases with n in 3..10, our mutants, on average, pass only 17.2% of the tests each, and for n in 4..10, pass only 12.5% of the tests. For n in {8, 9, 10}, there is no mutant that passes our contract. Any one of these three test cases kills all mutants, so

MT would consider a test suite with a single test case (for n = 8, n = 9 or n = 10), to be a mutation-adequate test suite, able to fully test this function, even though sqr5 is an integer-truncated quadratic function in n = -10..10.

Jia & Harman [44] define dumb mutants as those mutants that **fail all tests**. Averaged over a wide range of experiments, Jia and Harman have 41% dumb mutants. This is an extremely high number, and yet, the remaining 59% of mutants may still be quite dumb. Consider these questions:

- How many mutants are constant-valued or otherwise essentially dumb, yet are not considered dumb because they pass a trivial baseline test?
- How many more of Jia & Harman's single-mutation mutants fail all but one test, or all but a few tests?
- What percentage of nontrivial tests (tests excluding trivial tests) do mutants pass on average?

If mutants are similar to programs written by competent programmers, and they differ from correct version by relatively few faults, on average, they should pass a high percentage of nontrivial tests.

In our test, we have only one dumb mutant out of eight mutants (12.5%). But our mutants are not very bright on average: Any mutant, on average, fails 69% of all tests (n = 0..10), and 82.8% of nontrivial tests n = 3..10 (where output = 0 is not a pass). Two of the eight single-mutation mutants always return 0, and yet, they pass the test case of (0, 0) and so are not "dumb mutants" as per Jia & Harman. Two-mutation mutants are worse; 10 of 16 are constant-valued. Clearly, few syntactic differences do not produce programs with "relatively few faults."

The presence of constant-valued and dumb mutants is a strong argument against considering each mutant equal in calculating (and therefore inflating, through dumb mutants) the mutation adequacy score. Inflated adequacy score gives a false sense of security.

Most importantly, MT adequacy score should never be 1.0 for an inadequate test suite. For sqr5, some test suites with a single test case each (for example, sqr5(10) == 20) were found mutation-adequate for sqr5. Clearly, a single point cannot possibly start to explain, predict, thoroughly test or fully exercise a function that is quadratic for n in -10..10. Any line or curve that passes through (10,20) would be considered correctly implemented by this MT-adequate test suite.

5.5 Semantic Mutation Testing

5.5.1 Semantic Mutation Testing (SMT) Injects Nondeterministic Semantic Faults

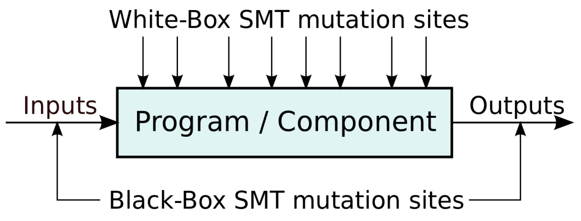


Figure 5.4: Black-box SMT only changes inputs and outputs, while white-box SMT may change the program or component at multiple sites.

Semantic mutation testing (SMT) is a fault injection method that introduces input/output (black-box) or internal (white-box) semantic state faults while a program P is being tested or verified against specifications. While the black-box variant does not change the program, white-box variant may change the program at multiple sites, as seen in fig.5.4. To make the faults small, we use nondeterministic random errors with a controlled standard deviation. To check issues of dependency on multiple errors, we inject faults in multiple locations, each with a very small probability, while assuring the behavior observed from outputs has changed. Compared to Untch's [78] meta-mutant mentioned above, we also create only one mutated program, but we do not selectively toggle mutation sites to leave only one active. During a single execution, each mutated location in our mutated program may introduce a small fault.

Analysis of syntactic MT examines effects of mutation on dynamic behavior of the program. DeMillo and Offutt's "necessity" condition [29] (and "infection", in Voas's PIE analysis [80]) holds when mutation/syntactic change causes a program state/data change. In SMT, we focus on necessity/infection caused by multiple mutations, but also attempt to limit the magnitude of program state change to have small semantic defects in our mutants.

For SMT, a mutant is not a mutated program, but rather, one recorded execution of the mutated program. Instead of verifying against a test suite as done in MT, we verify the mutant execution using the verifier. As we have a single randomized mutant with possibility of multiple concurrent mutations, each execution can be quite different in behavior.

SMT adequacy score of a verifier that passes P is the number of mutants (mutated executions; mutant test cases) not killed by the verifier divided by the number of non-equivalent mutants. Verifiers that fail P are disqualified, and get a score of 0.0. Due to randomized nature of mutated program, we need to run a large number of tests (multiple times per test input) to get a stable adequacy score.

Table 5.2 compares MT against two variants of SMT, black-box SMT (SMTb) and white-box SMT (SMTw). Note that, SMTb does not mutate the program at all, and SMTw creates a single mutated version of the program. SMT creates a score based on number of mutant test cases that the verifier passes.

Even though equivalence of arbitrary programs for all possible inputs is undecidable, equivalence for one mutated execution (mutant) is easy to check: If the output of mutant M_i for the test case T_j is the same as the output of P, mutant is equivalent, and could not possibly be killed by any specification/verifier. This means, unlike syntactic MT adequacy score, SMT adequacy score can always be calculated.

Table 5.2: Comparison of traditional Mutation Testing (MT) against black-box SMT (SMTb) and white-box SMT (SMTw). Here, we use == comparison for data, but equivalence of programs under all inputs is shown with ≡, and is generally undecidable.

Method	MT	SMTb	SMTw
Mutated Prog	M_i (**mutant**)	P (original prog)	M (composite)
Passed Case	$P(I_j) = O_j$	V(I_j, O_j)	V(I_j, O_j)
Mutant Case	O_j' = $M_i(I_j)$	I_j' = mutated I_j O_j' = P(I_j') O_j" = mutated O_j $<I_j$, O_j"$>$ is **mutant**	O_j' = M(I_j) $<I_j$, O_j'$>$ is **mutant**
Test	O_j == O$_j$' ?	V(I$_j$, O$_j$") ?	V(I$_j$, O$_j$') ?
Equivalent?	$M_i \equiv P$? (undecidable)	O$_j$" == P(I$_j$) ?	O$_j$' == P(I$_j$) ?

5.5.2 White-Box SMT

In white-box semantic mutation testing, errors can be introduced at any place where Java stack machine has values in its stack (expression stack). For example, the simple expression a + b - c has five spots for mutation: When local variables a, b, and c are read, and when expressions (a+b) and (a+b-c) are calculated. All five of these values may be mutated at the same time, but the probability of each mutation being nonzero is quite small.

In white-box SMT, a single composite mutated program ("meta-mutant") is generated to introduce random errors in program dynamic state. In our current implementation, this is done by Java bytecode manipulation (done only once per class), and does not require recompilation of class. Each execution with the same inputs gives us a slightly different test case. Our mutated program is nondeterministic, and cannot be equivalent to the original program so long as it contains at least one reachable mutation. In SMT, we do not talk about mutant programs, but rather, mutant test cases. A mutant test case is defined by inputs and the mutation decisions made in one execution of the program.

As there are many sites for mutation, if the probability of state change per mutation site is not very small, even a short program can quickly accumulate a large number of mutations (especially within loops) and deviate significantly from the original program's semantics.

We control the standard deviation of our random value generator to limit the deviation in program state space.

5.5.3 Data Flow Analysis for Crash Prevention

Sometimes, even though mutations keep the syntax of the program intact, they may cause significant semantic problems. Our experiments revealed that program crash caused by array index out of bounds exception is a common problem when random errors are introduced in integer values in programs. Similar to deadlock handling, we can logically prevent, dynamically avoid, or detect and recover from such critical faults.

We currently use crash prevention, which requires some static data flow analysis. Even though data flow analysis is a white-box approach, there are advantages to using it even while doing black-box fault injection. Alternatively, to dynamically avoid or recover properly, we would need to instrument the implementation (to check and fix every array index access, for example). Traditional MT employs what could be termed the "ostrich approach", of ignoring the problem, letting mutants cause crashes.

Using Java bytecode manipulation library ASM, we perform a data flow analysis, to discover when an int type data source (variable or value) is used directly or indirectly for indexing in an array. We discover all possible execution paths and track definition-use paths that may pass through various positions of the JVM expression stack. We refrain from mutating values from data sources that affect array indexing, to prevent array index crashes. In the example below, we will not mutate n and k as a single extra increment of k could cause a crash:

```
for(int i = 0; i < n + k; i++)
    sum += a[i];
```

Prevention is a static analysis approach, keeping execution safe without dependence on actual runtime values. This is more efficient compared to dynamic detect-and-recover approaches, but may be overly cautious at times. While preventing crash, this also eliminates potentially safe array size and cross-element mutations within the array. As we have done in our experiments, such mutations can separately be added with array-size-aware array mutation operators. Our approach can also be extended to handle other less-frequent causes of crash and state corruption, such as division by zero and using a type beyond its valid range.

5.5.4 Black-Box SMT

Black-box SMT only introduces errors to the inputs and the outputs of the program. To make sure this introduces a semantic change, the ideal approach is to mutate inputs, run program P with some form of crash-handling, and mutate outputs (possibly generate multiple variants for the same inputs). This time, the mutated inputs and outputs define a mutant; they represent the mutated execution. A verifier with a very high degree of concordance with implementation should kill (fail) most mutants.

Black-box SMT is closely related to interface MT [27] which deterministically and syntactically changes the interface elements, for example, by always incrementing one argument before method call. The main distinction from interface MT is that mutations in black-box SMT are nondeterministic, error is equal to 0 with some/high probability, and when nonzero, error is small in magnitude with high probability.

Specification mutation testing [12] also considers program as a black box, but instead of mutating program's outputs, it mutates the specifications themselves. If our verifiers are directly generated from specifications, this would correspond to mutating the verifier.

Black-box error injection is also used in fuzz testing [61] . Fuzz testing focuses only on stress-testing the implementation for robustness (avoiding crashes) whereas black-box SMT does not test the target program itself, but rather attempts to measure functional specification adequacy. We want to avoid, not discover, cases that cause crash.

Even in black-box SMT testing, if inputs are being mutated, we still recommend where code or bytecode is available, a data flow analysis to prevent array index crashes. In the example from the previous section, if n and k are program inputs, we should not mutate them.

5.5.5 Summary

For field-tested software that is evolving, specifications may be missing or outdated, and may need to be discovered or compared against the implementation. Incompleteness of specifications is a common problem that is hard to discover or fix. This can cause specifications and automated verifiers based on them to continually verify an incorrect implementation. Not having up-to-date specifications hinders productivity and quality of common software development tasks such as verification, maintenance, comprehension and reuse.

The central question of specification-implementation concordance is “How faithfully (accurately) does this specification represent the behavior of this implementation?” An automated verifier is really a machine language form of requirements specification, so this is also the central question of verifier-implementation concordance.

In section 5.2, we have defined a subsumption relation for functional specifications and verifiers. We have shown that subsumption relation we have defined forms a lattice, a partially ordered set where any two elements have a supremum (least upper bound, “join”) and an infimum (greatest lower bound, “meet”). After considering the two extremes, top and bottom elements that are always false and always true, we defined complete verifier V_P for a program P that completely specifies its behavior.

Using our subsumption relation, we stated our requirements for a good verifier adequacy score:

1. $score(P_0, \bot_{\mathbb{I}\times\mathbb{O}}) = 0.0$
2. $pass(V_{P_0}, P_0) \wedge \neg\exists V : (V > V_{P_0} \wedge pass(V, P_0)) \Longrightarrow score(P_0, V_{P_0}) = 1.0$
3. $V_1 > V_2 \wedge pass(V_1, P_0) \Longrightarrow score(P_0, V_1) > score(P_0, V_2)$
4. $fail(V, P_0) \Longrightarrow score(P_0, V) = 0.0$

Note that we do not have a requirement about how high or low the adequacy score values need to be so long as an improvement in verifier (more strict verifier that fails a larger number of candidate programs while passing our program P_0) causes an improvement in the adequacy score. Any strictly monotonic mapping would preserve these qualities.

Even though our specification adequacy metric resembles test suite adequacy metric, we cannot use any of the code coverage metrics as stated earlier and shown with an example in section 5.3.

In section 5.4, we have shown that traditional mutation testing (MT) has a number of serious shortcomings:

- Detecting equivalence of mutants is in general undecidable, and in practice prohibitively time consuming, and even DeMillo in his seminal work in 1978 quietly skipped this step.

- Without knowing which mutants are equivalent, mutation adequacy score cannot be calculated.

- MT has high computational complexity.

- MT injects small syntactic faults that are hard to notice in code review, but which often cause major semantic differences that are easy to notice in program behavior.

- Many mutants (single syntactic change) fail all test cases (41% of all mutants in 10 programs Jia & Harman tested in [44]).

- Very small syntactic error in source code may cause major semantic difference in behavior; mutants that do not fail all test cases may still fail a large percentage of test cases.

To address these problems, we recommend controlling the magnitude of semantic errors introduced. Our semantic mutation testing (SMT) approach introduces small-probability small-magnitude random error in the values of program's variables that collectively constitute program state.

SMT has two variants, white-box SMT and black-box SMT. White-box SMT injects errors in multiple sites of program code by using a single composite mutant program that introduces nondeterministic small-probability small-magnitude error terms in the statements and expressions of program code. Black-box SMT only introduces errors in inputs and outputs, and does not need program code to be mutated.

SMT introduces random errors, the behavior is nondeterministic, and repeated runs with same inputs may behave quite differently. To prevent error injection from causing a crash, we perform data flow analysis and avoid injecting errors that may cause crashes.

Unlike MT, for SMT we consider one recorded "mutated run" with error to be one "mutant". This makes equivalence very decidable: If original unmutated program also gives the same output as our mutated run, for the same inputs, the mutated run is equivalent and should not be considered in the calculation of mutation adequacy score. This equivalance test can be performed automatically, and therefore, unlike in MT, SMT adequacy score can always be calculated, and can be calculated without human intervention.

In the next chapter we will evaluate our method on a number of test programs with alternative specification/verifier candidates.

Chapter 6

SMT Experiments

6.1 Introduction

As introduced in the previous chapter, Semantic Mutation Testing (SMT) is suggested as a fault-injection method that avoids some of the problems we observe with traditional mutation testing, by controlling the magnitude and frequency of errors. SMT has two variants, Black-Box SMT and White-Box SMT. In this chapter, through some short functions and alternative specifications, we demonstrate use of SMT adequacy metric for a specification (and corresponding automated verifier). Note that this is performed without human intervention, by using mutated versions of the program to find out if alternative verifiers can distinguish between correct and faulty behaviors.

Our implementation uses intermediate language (Java bytecode) manipulation to avoid need for recompilation. In our earlier tests, we quickly discovered that mutants (mutated versions of the programs) often crash due to "array index out of bounds" exception. We use automated data flow analysis to avoid most cases of array index out of bounds by not mutating values used in indexing.

We describe alternative specifications using Design by Contract (DBC) [60]. In our experiments we avoid the need for preconditions by allowing any non-null value to be valid at input. This leaves only the postconditions, which fully specify what each method must accomplish.

Our DBC specifications do not need to check for preserving object invariants because our target components are side-effect-free static methods, and there is no "this" object. [1]

Even though we use DBC, our results do not depend on this choice of verifier implementation. For each alternative specification of each method, the postcondition really defines what the method is expected to accomplish. Any type of verifier that correctly implements these specifications would produce the same set of adequacy score values.

6.2 Experiments 1/2: One Method, Seven Alternative Verifiers/Specifications

6.2.1 Sorting, Alternative Specifications

In our first set of experiments, we tested traditional syntactic mutation testing, white-box and black-box semantic mutation testing on a sorting function with seven alternative specifications. The sort method, sort(int[] ar), takes an integer array, and returns a nondecreasing sorted version of the integer array.

Seven alternative contracts (our DBC specifications) are ordered in our approximate completeness/strictness order in table 6.1; actual order depends on input pattern frequencies. Here, bag(.) converts the collection (array) to a bag/multiset.

C_X is too strict: It will not allow duplicate elements in the returned array, so it fails even correctly implemented sort method for 54% of our test cases. Due to this, its killed mutants score is erroneously inflated. Mutation testing disqualifies C_X for failing unmutated

[1]As far as SMT is concerned, a "this" object is just another input to a function that may be mutated. Even though hidden in OO programming languages, this first argument would be explicit in a C program that uses an Abstract Data Type (ADT). For example, in our second set of tests, the four functions that each take an integer array could instead be four no-argument methods in an ArrayAnalyzer class that takes an integer array in its constructor.

Table 6.1: Seven alternative contracts for sort method

Contract	Postcondition
C_0. NO_TEST	true
C_1. SAME_LEN	\$result.length == ar.length
C_2. SORTED	∀ i: \$result[i] <= \$result[i+1]
C_3. SAME_VALUES	bag(\$result).equals(bag(ar))
C_4. SORTED_LEN	SAME_LEN && SORTED
C_5. SORTED_VALUES	SAME_VALUES && SORTED
C_X. SORTED_NOTEQ	∀ i: \$result[i] < \$result[i+1]

program.

6.2.2 Traditional (Syntactic) Mutation Testing

We used Jumble [42] to run traditional mutation tests on this method. Jumble uses Java bytecode rather than source code manipulation to eliminate the need for recompilation per mutation. By using its own classloader, Jumble can reload mutated versions of classes that are being tested, without need to restart or spawn a new JVM for each class that must be tested. Jumble can be used with Java versions 1.4, 1.5, 1.6, and it can analyze JUnit 3 and JUnit 4 test classes to detect and run individual test cases separately. Jumble was being used in 2007 in a continuous-test of a 370,000-line Java software every 15 minutes [42] . We ran Jumble with standard settings.

Unfortunately, traditional MT with Jumble did very poorly on the original code, as seen in table 6.2, due to finding many mutation points in the unrelated debugging/logging print statements. Even though debug option was turned off and these statements never ran in unmutated original program, adequacy scores were very poor due to our unit test not discovering changes in the debug/log code. Of the 17 mutation points found, 11 mutation points were in debugging code that never got executed.

In order to prove MT method can subsume code coverage criteria, MT researchers have added and continue to use degenerate "mutation" operators ([66] [4]) that deviate from the original purpose of making mutation operators represent reasonable errors competent programmer may make. One such example is “reachability” test that passes an MT test unconditionally when a point in code is reached. These degenerate mutation operators that fail or pass without considering the verifier cannot measure verifier adequacy, and therefore they degrade the MT score when MT is used for verifier adequacy.

In our experiments, two mutation points out of eleven were supposedly even caught by no-test unit test, probably due to such reachability test mutation operators. This left a compressed and misleading range of adequacy score values that gave 0.11 to the no-contract worst case and 0.35 to the best contract, instead of using the full range of 0.0 to 1.0.

After we edited our code to manually remove all debugging/logging code, we were left with only six mutation points, of which two again were supposedly caught by the no-test contract. The adequacy score is erroneously nonzero for C_0. Actually, in both original and edited code, there are only two possible adequacy scores (minimum and maximum possible scores). As no interim values exist, the contracts are not really distinguished by their scores.

Table 6.2: Traditional (Syntactic) Mutation Test results. C_X gave error "test class is broken", and was disqualified, as it failed the original program.

Contract	Mutant Adequacy Score	
	Original Code	Edited Code
C_0. NO_TEST	0.11	0.33
C_1. SAME_LEN	0.11	0.33
C_2. SORTED	0.35	1.00
C_3. SAME_VALUES	0.11	0.33
C_4. SORTED_LEN	0.35	1.00
C_5. SORTED_VALUES	0.35	1.00
C_X. SORTED_NOTEQ	–	–

In traditional syntactic mutation testing, any mutation is considered equivalent, regardless of whether it is reachable or how many times it gets executed. These tests demonstrate that considering every mutation point equal gives misleading adequacy scores.

6.2.3 Black-Box Tests

We randomly produced 1000 input-output sets using our correct implementation of sort method. After skipping trivial cases of arrays of size 0 and 1, we were left with 803 input-output sets. We mutated the returned sorted array using four array mutation operations:

- **swap**: swaps two elements of array ar
- **replace**: replaces an element of array ar with a randomly picked element from another array of similar values
- **resize**: either duplicates an element of array to grow the array by one element, or removes one element to shrink the array by one element.
- **random**: replaces an element of array ar with a random int value

For each of the 803 input-output sets, we produced 200 mutated versions of the output array using 1 to 30 mutations per mutant, for a total of 160,600 mutant test cases. Comparing with correct output from unmutated program, we found 146,179 of these to be non-equivalent mutant test cases. Table 6.3 shows our test results.

This may sound like a large number of test cases, but considering a large number of test cases allows us to fully exercise the program and the verifier. In SMT, we are not trying to evaluate the test input generator or the richness of inputs, which is the only focus of code coverage, and partial focus of MT. Using a rich set of inputs makes our score evaluate only the adequacy of the verifier. Unlike in mutation tests where the number of test cases is decided by the program code, SMT also allows us to decide how many tests we prefer to run. In SMT, we have the choice to adjust the number of test cases to provide better coverage or faster completion of tests.

Recall that C_X is too strict to even accept correct implementation; this behavior inflates the number of mutants killed by C_X, as it even kills some equivalent mutants. In this case, the ordering of semantic mutation adequacy scores is as we expect; it corresponds to our prior belief of how complete each specification is. As C_5 is a complete specification (modulo invariants), its mutation adequacy score is 1.0.

Table 6.3: Black-Box Semantic Mutation Test results. C_X is again disqualified.

Contract	Mutants Killed	Adequacy Score (Proportion)
C_0. NO_TEST	0	0.00
C_1. SAME_LEN	82,245	0.56
C_2. SORTED	103,385	0.71
C_3. SAME_VALUES	125,597	0.86
C_4. SORTED_LEN	132,531	0.91
C_5. SORTED_VALUES	146,179	1.00
C_X. SORTED_NOTEQ	141,838	– (0.00)

6.2.4 White-Box Tests

We ran our data flow analysis to prevent mutation of values that may cause array index out of bounds exception, using any array access and new array creation operations as our targets. For example, whatever value or variable reaches top of the JVM expression stack for an IALOAD (int array load) instruction should never be mutated. At all remaining sites of integer value operation in the JVM stack machine, we modified the compiled Java bytecode to insert a call to our stateless mutater method to inject error to the int value on top of the expression stack. We did not need access to source code or recompilation to produce this composite randomized mutant.

The amplitude and frequency of mutations can be numerically adjusted in semantic mutation testing. We used integer-rounded Gaussian distributions as our additive error terms, with two different standard deviations; $\sigma = 0.2$ and $\sigma = 0.5$. Values in (-0.5, 0.5) get rounded to 0 and do not cause any state change. This happens 98.76% of the time with $\sigma = 0.2$, and 68.27% of the time with $\sigma = 0.5$. These two tests are significantly different; the $\sigma = 0.5$ case has nonzero error added about 25.6 times more often than the $\sigma = 0.2$ case.

The results for both values of standard deviations are shown in table 6.4. For tests with

$\sigma = 0.2$, 812 non-trivial cases with 200 mutant runs each gave us 162,400 test cases, of which only 7,095 were non-equivalent. For tests with $\sigma = 0.5$, 787 non-trivial input-output sets each with 200 mutant runs gave us 157,400 test cases, of which 102,868 were non-equivalent. In both cases, the faulty contract C_X failed more mutants than there were non-equivalent mutants.

As we do not mutate int values that are directly or indirectly used in array indexing, the length of the array never changes, and two pairs of contracts that differ only in checking array length produce exactly the same values: C_0 and C_1 both kill (fail) no mutants, and C_2 and C_4 always kill the same number of mutants.

Our metric suggests that sortedness of output (C_2) is easier to satisfy than preserving the bag of values from the input array (C_3). This is understandable, as any one mutation to any of the values during the execution will always change the bag of values, but small enough changes in values may not change the ordering of values, causing the output array to remain sorted while having a different bag of values.

Table 6.4: White-Box Semantic Mutation Test results. C_X is once again disqualified.

Contract	Mutant Adequacy Score	
	$\sigma = 0.2$ (7,095 $M_i \not\equiv P$)	$\sigma = 0.5$ (102,868 $M_i \not\equiv P$)
C_0. NO_TEST	0.00	0.00
C_1. SAME_LEN	0.00	0.00
C_2. SORTED	0.37	0.47
C_3. SAME_VALUES	0.86	0.92
C_4. SORTED_LEN	0.37	0.47
C_5. SORTED_VALUES	1.00	1.00
C_X. SORTED_NOTEQ	–	–
	(117,442 killed)	(109,729 killed)

6.2.5 Discussion

We have shown feasibility of measuring specification quality by how often specification fails programs with small semantic errors. Compared to traditional (syntactic) mutation testing, both our white-box and black-box semantic mutation testing approaches produce adequacy scores that better represent the concordance of the specification with our correct implementation. Our data-structure-aware black-box mutation operations gave better results compared to primitive-value-aware single mutation operator for integer values injected into method implementation (white-box).

Our crash prevention avoids any mutations that could have an effect in array sizes and which element is accessed. Other mutation operators such as the array-mutation operators used for black box testing can also be introduced to accessed arrays, after proper data flow analysis to prevent array index crashes. As always, the mutation operators will be most useful if they represent common types and patterns of faults.

Our experiments show that both variants of our method (white-box and black-box SMT) perform better than traditional (syntactic) MT in evaluating quality of specifications as compared to an implementation.

6.3 Experiments 2/2: Four Methods, Five Alternative Verifiers/Specifications Each

6.3.1 Introduction

Using four simple target programs, with five alternative verifiers each, we compare MT, white-box SMT and black-box SMT. In our experiments, SMT performs to our expectations for an adequacy measure of automated test verifiers, and adequacy of the functional specifications implemented by these verifiers.

We used the first three of our four requirements for verifier adequacy score listed in section 2.3 to judge the quality of different adequacy measures. As we do not have any verifiers that fail the original program, we did not use the fourth requirement that such verifiers get adequacy score of 0.0 (or terminate with an error message), even though all methods used do this, so long as there is at least one test input that causes verifier to fail the original program.

6.3.2 Target Programs and Alternative Specifications

We conducted three types of mutation testing experiments, on four small target functions, each with five alternative verifiers in the form of DBC contracts.

Three types of mutation testing we used were:

- J. MT: Jumble (traditional, syntactic MT)
- B. SMT-B: Black-box SMT (changes inputs/outputs only)
- W. SMT-W: White-box SMT (changes bytecode)

Our four target methods (functions) each take an integer array. The integer array arguments can be any size, including empty (`new int[0]`), but should not be null. The four target methods are:

- F_1: findFirstDuplicate Returns the first duplicate element's index, or -1
- F_2: findRange Finds the min and max values in the array; returns {0,0} for {} (empty array).
- F_3: getIntStats Calculates number of elements, their sum, and sum of squares.
- F_4: getStats Calculates the mean and standard deviation of values in the array.

F_2-F_4 return multiple values. In the following, these functions return primitive values and arrays instead of value objects, so as not to introduce more classes and structures. Our implementation can analyze and process both arrays and value objects, so composite types with different values can also be returned.

- F_1: `int` : index
- F_2: `int[2]` : <min, max>
- F_3: `int[3]` : <n, sum, sumOfSquares>
- F_4: `double[2]`: <mean, stdDev>

For F_3, n holds the array length, and for F_4, stdDev holds the standard deviation.

The five contracts (our verifiers) are ordered in our opinion of how strict they are, for each function:

- C_0: Empty contract
- ...
- C_4: Full, complete contract

First contract C_0 always returns true, and accepts any result returned by these methods. The four methods and the 16 non-trivial contracts for them are shown in table 6.5. We will use dotted notation to talk about a specific contract; for example, $F_1.C_3$ refers to the isDup contract for findFirstDuplicate method. Contracts are not comparable between functions; so we cannot say $F_1.C_3$ is similar to $F_2.C_3$ in strictness, and therefore, their scores need not be similar in value either.

Notes, observations:

1. Our contracts are in perceived order of strictness; for average nontrivial arrays, we expect earlier contracts to be easier to satisfy than later ones.

2. Some contracts strictly subsume others. Table 6.6 shows that of 16 possible comparisons of C_{i+1} versus C_i, 11 of them are strictly ordered. For F_2, we have $C_4 > C_3 > C_2 > C_1 > C_0$.

3. Whatever is not strictly ordered is ordered by the probability of accidentally getting the value correct, which is generally harder for values selected from a larger set of possible values. Thus, in F_3, sumOfSquares (C_3) is harder to get correct compared to sum (C_2), which in turn is harder to get correct compared to size (C_1). As stated before, the actual order discovered by a method may differ from this.

Table 6.5: Four target functions, five alternative contracts each.

Functions & Contracts	Description
C_0. true (all fns)	accepts any result; always returns true
F_1. findFirstDuplicate	Returns "ind", the first duplicate element's index, or -1 if there are no duplicates.
C_1. validIndex	ind is a valid index for array, or ind is -1
C_2. noDupsBefore	There is no duplicate in ar before index ind
C_3. isDup	ind is -1, or ar[ind] is actually a duplicate
C_4. isFirstDup	ar[ind] is the first duplicate in array ar ($C_4 \equiv C_2$ && C_3)
F_2. findRange	Returns the min and max values in the array; returns {0,0} for empty array.
C_1. min_LE_max	min $\leq$ max
C_2. first	`ar[0]` is in min..max
C_3. firstLast	`ar[0]` and `ar[len-1]` are in min..max
C_4. range_correct	(min, max) are correct min & max values for ar.
F_3. getIntStats	Returns array size n, sum & sumOfSquares for array ar.
C_1. size	n is correct; n $\equiv$ the actual size of the array (len).
C_2. sum	sum is correct; sum $\equiv$ `ar[0]` + ... `ar[len-1]`
C_3. sumOfSquares	sumOfSquares is correct; it is the sum of all `ar[i]*ar[i]`
C_4. all_correct	n, sum and sumOfSquares are correct ($C_4 \equiv C_1$ && C_2 && C_3)
F_4. getStats	Returns double-valued mean and stdDev for ar.
C_1. mean_GE_min	mean $\geq$ min value in array ar
C_2. mean	mean is correct; mean $\equiv$ sum(ar)/len
C_3. stdDev	stdDev is the actual standard deviation of ar.
C_4. both_correct	Both mean & stdDev are correct ($C_4 = C_2$ && C_3).

Table 6.6: Strict subsumption between our contracts for functions

Function	$C_1 > C_0$	$C_2 > C_1$	$C_3 > C_2$	$C_4 > C_3$
F_1	T	F	F	T
F_2	T	T	T	T
F_3	T	F	F	T
F_4	T	T	F	T

6.3.3 Test Inputs

Our four functions each take one integer array as input. To test every method with the same set of inputs, we generated 100 input arrays with sizes from 0 to 49 elements (with median size 15, to have various smaller tests), where the values in the arrays were randomly selected from -10 to 20, so that negative values are also tested and there is a nonzero chance of duplicates even in smaller arrays.

6.3.4 Black-Box SMT Implementation

Our inputs are randomly generated. We focused only on output mutations in our black-box mutation experiments. For each output term (dimension), we used the same standard deviation value equal to 3.0 divided by the number of output terms. For each integer output, we used added error terms using integer-rounded Gaussian distribution in our tests. This makes sure that there is a large probability (68.27%) of adding no error at a given time. We repeatedly generated error term vectors for outputs and discarded nonzero vectors for our tests. For F_4 with two double-valued outputs, we used a prior probability of 50% (as there are only two outputs) of any output error term being 0.0. Again, we discarded the zero error term vectors.

6.3.5 White-Box SMT Implementation

We also tested these methods and contracts with white-box SMT, as we have done in 6.2.4. Our white-box mutater implementation currently only mutates integer values. Even though the last function, F_4 (getStats), returns two double values, these values are calculated from integer values in an integer array, so mutating only integers will still cause output to be mutated.

As we do not have whole array mutation operators, array size cannot change. As we avoid changing index variables, the return value of F_1 (findFirstDuplicate) is only changed indirectly, by changing integer values read in array, which can change the first index of a duplicated value in the array. But because the array size does not change and index cannot become valid, we cannot expect $F_1.C_1$, which checks for valid index value, to ever fail under SMT-W, and therefore this contract should get score 0.0 and appear to be equivalent to empty contract $F_1.C_0$ that also never fails.

A similar situation arises with F_3 (intStats) where C_1 checks whether size is correct. As array size cannot change, and a variable that holds array index or size should not be mutated to avoid crashes, $F_3.C_1$ should never fail under SMT-W, and should therefore get score 0.0 and appear equivalent to empty contract C_0 which also never fails.

6.3.6 Traditional MT With Jumble

As before, we used Jumble with standard settings to run traditional mutation tests.

For Jumble, the granularity of adequacy score depends on the number of mutants for the program. Each mutant is run against each test. We did not observe any difference in adequacy scores when we used 1000 or 10000 similarly generated tests instead of the 100 we have used.

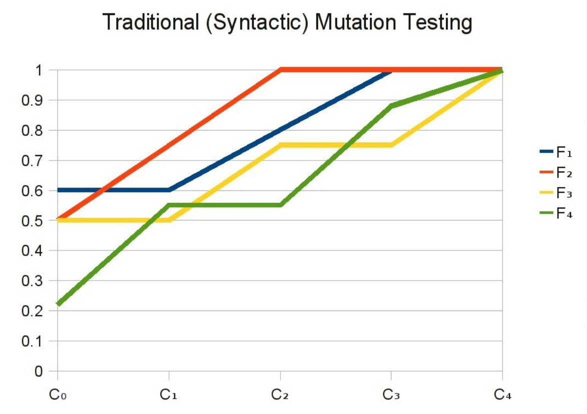


Figure 6.1: Traditional (Syntactic) Mutation Testing results using Jumble

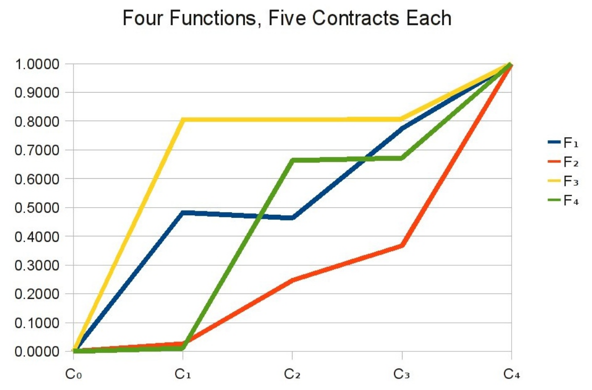


Figure 6.2: Black-Box Semantic Mutation Test results

6.3.7 Results & Observations

Fig.6.1 shows the traditional mutation testing results. We notice some problems:

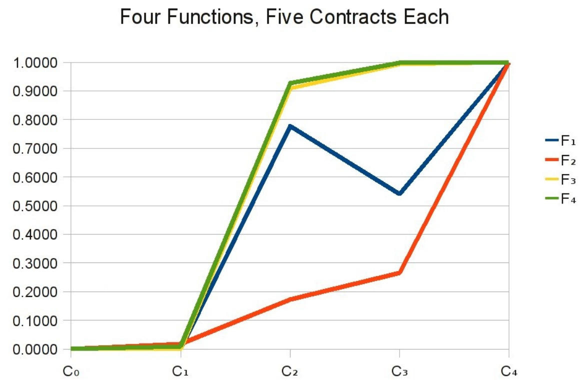


Figure 6.3: White-Box Semantic Mutation Test results

- C_0 (always true) checks nothing, should get score 0.0, but it never gets 0.0 (probably due to code-coverage emulation in mutation testing, with "trap_on_statement" and similar mutations, which verify the test input generator, but not the verifier)

- For all functions, $C_1 > C_0$, but for F_1 & F_3, they get the same MT adequacy scores. According to this, $F_1.C_1$ and $F_3.C_1$ are equivalent to having no checks.

- For all functions, $C_4 > C_j$, for any $j < 4$, so no other contract should get 1.0 (and be considered a mutation-adequate test suite). But for F_1, C_3 gets 1.0, and for F_2, both C_2 and C_3 get 1.0. $F_2.C_2$ only checks if first element of the array is in [min, max]. Any values of min and max with min $\leq$ ar[0] $\leq$ max would satisfy C_2.

- For F_4, $C_2 > C_1$, yet C_1 and C_2 get the same score. C_1 only checks mean $\geq$ min(ar), which would be satisfied for any wrong value of mean with $\exists i : mean \geq ar[i]$. C_2 checks that mean holds the correct value, which is much more strict than C_1.

Fig.6.2 shows the black-box semantic mutation testing results. We notice that C_0 always gets score 0.0, C_4 always gets score 1.0, and no other contract is considered equivalent in strength to these extremes. We do notice a few problems, though:

- For F_1, C_2 gets a lower (almost same) score compared to C_1.

- For F_3, C_1, C_2 and C_3 seem just as easy to satisfy, because in black-box semantic mutation testing, it is just as likely to mutate n (=size), sum and sumOfSquares values returned, but these are increasingly more informative, and should be harder to satisfy.

- For F_4, C_2 and C_3 get same score, similar to F_3, with mean and stdDev values as likely to be mutated but stdDev (due to squared values) holding more information and harder to get correct. In practice, both are quite hard to randomly get correct, and the difference is small.

These are perceived order of strictness; there is no strict subsumption relation between these contracts/verifiers, the order can change with implicitly presumed probability distributions and importance/weight of different possible input values.

Fig.6.3 shows the white-box semantic mutation testing results. We notice that most contracts are ordered properly. We also notice the shortcomings of not having array mutation (therefore never changing array size) in our implementation of white-box semantic mutation testing. Because of not changing array size and not mutating array index (and array size) values and variables, we have:

- For F_1, $C_1 = C_0$ with score 0.0. Index returned by findFirstDuplicate is always valid even in mutated program. We cannot make array index variable not have a valid index as this is exactly how our preliminary tests have crashed with ArrayIndexOutOfBoundsException.
- For F_3, $C_1 = C_0$ with score 0.0 again, as array size values and variables are never mutated.
- There are also reversed ordering in some other scores, but these are, as before, always for contracts with perceived order of strictness rather then actual subsumption.

We can see that black-box mutation testing, because it does not have access to computational complexity or statistical knowledge that could be gathered over time, considers elements of output with same interface complexity (held in type structures of similar complexity) to be equivalent in value.

We believe black-box mutation testing can be further improved in the case of these not strictly comparable contracts, by using stateful mutation operators that depend on the universe of observed/possible values generated by methods. In this way, we hope to overcome its shortcoming of considering any output element to be as informative (and as hard to get correct) as any other output element.

Traditional mutation testing produces sets of mutants of size approximately proportional to code size. More complex computations generally have large impact on adequacy score, except computational complexity of loops are often not represented. Most sets of mutation operators do not contain the rather complex stateful mutation operators that check for number of times a loop executes (the SMTT and SMTC "multiple trip trap/continue" operators defined in [4]). Consequently, the number of times a loop executes does not change the score so long as test suite exercises every branch.

Table 6.7 compares traditional MT against black-box and white-box SMT on the first three of the four requirements for proper verifier adequacy score mentioned in previous chapter . We see that MT never produced 0.0 for empty contract, and rewarded strictly better (subsuming) next contract with higher adequacy score less than half the time in our experiments. In comparison, Black-box SMT got a perfect score: It always gave 0.0 to C$_0$, always gave 1.0 to C$_1$, and always caused strict subsumption between contracts to be noticeable by a difference in adequacy score: Strictly better contracts always got better adequacy scores. Our implementation of white-box SMT does not use array mutation operators, and this causes in two cases (F$_1$ and F$_2$) the first contract C$_1$ to get score 0.0 and be indistinguishable from C$_0$. In all other cases of strict subsumption, SMT-W has produced a strictly better score.

Table 6.7: An evaluation of traditional MT and SMT on requirements defined in chapter 5.

Method	$C_0 \to 0$	$C_4 \to 1$	$C_{i+1} > C_i \implies$ $score(C_{i+1}) > score(C_i)$	Average Score
MT	0/4	4/4	5/11	
	0.0	1.0	0.4545	0.4848
Black-Box SMT	4/4	4/4	11/11	
	1.0	1.0	1.0	1.0000
White-Box SMT	4/4	4/4	9/11	
	1.0	1.0	0.8182	0.9394

6.4 Conclusions and Future Work

6.4.1 Conclusions

We have introduced a method, semantic mutation testing (SMT), to evaluate the quality of automated test verifiers (oracles) and to measure verifier-implementation concordance. To the degree that the implementation itself is tested and known to conform to user's needs, our semantic mutation specification adequacy score also measures how well the specifications match actual user requirements.

None of the existing code coverage criteria can be used for this problem, as input variation that fully exercises any code coverage may be coupled with an empty contract that checks nothing, but would get full code coverage score. In our experiments, we found that semantic mutation testing gives results that mostly conform to our expectations of how contracts (our executable specifications) are ordered.

Using four functions and five alternative DBC contracts each, we analyzed the strengths and weaknesses of SMT, and compared black-box and white-box variants of SMT against traditional mutation testing, MT. We found MT to be inadequate, and both variants of SMT to be very adequate in:

- Discovering when a contract is equivalent to empty contract that passes any program; such contracts get score = 0.0.
- Discovering when a contract is correct and complete; such contracts get score = 1.0.
- Discovering when one contract strictly subsumes (and is therefore better than) another contract, by producing a better adequacy score.

6.4.2 Future Work

As an alternative to randomly created test cases as seen in our experiments, we can use any existing test suites, or consider gathering test data in situ, by saving input-output sets (using serialization for reference types, objects) from a component while the software system is running.

We believe this simple-to-compute measure of specification-implementation concordance can help automate measuring quality of suspected-to-be-outdated as well as rediscovered or competing specifications. We will use this specification evaluation method in guiding testers in discovering lost specifications and maintaining/evolving outdated specifications through live sessions with our VERDICTS ("Visual Exploratory Requirements Discovery and Injection for Comprehension and Testing of Software") research implementation.

Source code and data from these experiments are available at our research page [16].

Chapter 7

Conclusions

7.1 VERDICTS

In the preceding chapters, starting from some common problem scenarios of software development, we have envisioned and implemented a tool to help partial automation in some software development tasks. We also proposed and tested a method to evaluate verifier adequacy.

Specifically, we have considered the common problems in areas of testing, verification, comprehension, debugging, specifications discovery and last but not least measuring automated verifier adequacy. We looked at standard debugging paradigm, and the many ways that it fails to support exploratory efforts, and how and why it may be worse than old-fashioned print statements. We also examined some innovative debugging methods, as well as other common approaches that help with program dynamic behavior analysis such as logging and tracing.

Our proposed method is VERDICTS, Visual Exploratory Requirements Discovery and Injections for Comprehension and Testing of Software. This method allows for rapid interaction with the target software, switching between verification/testing and analysis/discovery modes of operation. To automate testing we used contracts that allow for rapidly specifying the requirements of the program's components. Various methods, technologies and techniques have been used to implement VERDICTS. These technologies and techniques were listed in Chapter 3, and how they work in VERDICTS has been explained in both chapters 3 and 4.

VERDICTS uses Design by Contract, aspect-oriented programming with AspectJ, interpreted Java with Beanshell and various statistical views to help the discovery of features and behaviors of the target program. One of the most important features of VERDICTS is that the target program does not have to be recompiled and its source code need not be modified (or even be available) for the user to add new tests and verifiers. Eliminating compilation delays allows the user to develop a mental model of how the program works using a rapid feedback cycle. This rapid feedback cycle also introduces new approaches for discovering requirements. Specifically, instead of having to discover the final complete version of the requirements, the user can evolve requirements specifications, mix requirements specifications with actual behavior specifications, and also add verifiers that are intended to fail merely in order to discover the requirements and program behavior.

Full-program-state-recording approaches such as Omniscient Debugging [58], Reversible and Bidirectional Debugging [17] and Whyline [48] often introduce a very large performance penalty, need for disk and memory space, and inability to naturally interact with the target program. In our tests, we found VERDICTS to be efficient, capable of recording thousands of method calls in a few seconds while allowing target program to remain interactive [13].

As we stated in the first chapter, we will avoid making any claims about improved comprehension at this time, mainly because comprehension tests and many other end-user tests of VERDICTS would best be performed with a large number of test subjects with varying levels of software development experience. Similar to researchers who create new types of visualizations, we present our VERDICTS method and leave it to readers to judge the utility.

Through various tests, we demonstrated that VERDICTS compares very favorably to a modern Java IDE that would likely be used for these tasks. For three innovative features of VERDICTS, on almost all metrics, in almost all tests, VERDICTS performs better or significantly better than Eclipse. On average, with VERDICTS, we could analyze 23 times more method calls per minute, and could verify our hypotheses for 11 times more method calls per minute (verify eight times more method calls in 28% shorter time).

7.2 Semantic Mutation Testing

Even though VERDICTS is a platform that is designed to allow for requirements discovery and rapid verification of software, it is not a fully automated system. Rather, it is a partially automated system that aims to help a user verify and discover behavior of programs while applying human intelligence and professional experience. While using VERDICTS, there may be multiple versions of verifiers and requirements for a program component. One situation arises when a verifier is evolved over time. Another situation may arise when documentation and behavior seem to conflict and two verifiers, one created to align with the documentation, and another created to align with the current behavior, may have to be compared against the program itself.

As we do not have a third-party arbiter that can decide whether the verifiers and requirements specifications adequately represent what the end uses require, the only way we can compare the quality of verifiers is by using the target program. What we are really measuring is "specification-implementation concordance" (and verifier-implementation concordance). Even though end users are not involved, this adequacy measure is of great importance for verification, comprehension, and various maintenance tasks. For component reuse, this is exactly the measure we need, to make sure the component delivers exactly what it promises.

For this purpose we have introduced a method called Semantic Mutation Testing (SMT). After briefly introducing the method in chapter 1 and stating that this method is an approach to measure automated verifier adequacy, in chapter 5 we have considered in detail whether existing methods can be used for this purpose.

The idea of using the program itself to measure adequacy of a verification method is not a new one. Various code coverage criteria exist for this purpose when test suites are used for verification. But as we have seen in fig.5.1, using a verifier requires a different approach from using a test suite; specifically we need a test input generator as a first stage and verifier as a second stage of verification. One way that this approach is different is that a test suite is a memorized set of inputs and outputs and merely represents a sample in the space of program inputs and outputs that collectively represent the program behavior. In contrast, a verifier can allow for a very flexible test input generator and the number of samples can be changed at the time of testing, allowing for a focus on higher coverage or more efficient verification.

When we looked at code coverage criteria, we quickly realized that all code coverage criteria completely ignore outputs and pass/fail judgement of test cases and verifiers. Code coverage criteria therefore only measure the adequacy of test inputs in terms of how inputs exercise the program. Another approach for measuring test suite adequacy, Mutation Testing (MT), is more promising, as it also considers and evaluates the program outputs. Even though all of these approaches use the program itself to measure the adequacy of testing, this does not mean that the test is merely run against the program. Both the code coverage criteria and traditional MT instead use source code analyzers and at times execution interception in order to examine the program instructions, to track execution of each instruction, or possibly to even modify the instructions to introduce an error in the program.

In chapter 5, after examining some serious shortcomings of traditional MT, we described our approach, SMT, in some detail.

The biggest problems of traditional MT have also been observed by other researchers. One problem that we have looked at in chapter 5 is concerned with one of the original observations of MT: “Programs developed by competent programmers vary from correctly implemented programs by only a few faults". Unfortunately, MT researchers generally accept this to mean that programs have only few syntactic errors, even though a small syntactic error in code often causes major semantic difference in behavior. In chapter 5 we have shown this to be true in a small example. Beyond our simple example, researchers have observed that there are many mutants that pass no test cases in the test suite. This means a single syntactic variation has modified the program from passing all test cases to passing none of the test cases. Such mutants are dubbed "dumb mutants", and they are rather prevalent in traditional mutation testing (41% in a large series of tests by Jia & Harman [44]). Beyond such dumb mutants, we have also observed in our small example that many mutants fail a large percentage of test cases.

Another significant problem with MT is that MT requires that we check semantic equivalence of mutants with respect to original programs. As we have noted in chapter 5, even in his seminal work on MT in 1978 when programs were much shorter than today, DeMillo quietly skipped semantic equivalence checking for mutants, even though there is no mention in his paper of this significant omission [28].

Without checking for semantic equivalence of mutants, MT adequacy score cannot be evaluated. This is compounded with the fact that MT generates poor mutants and considers every mutant to be equally valuable in calculating the adequacy score, which causes an inflated adequacy score.

SMT, in contrast, focuses either on modifying the inputs and outputs of the program, or modifying the values in expressions within the program. The error magnitude is controlled, and as variable values collectively represent the state of the program, state and program behavior is often modified only in a small way rather than changed significantly. SMT aims to ensure that small faults are introduced in the behavior of the program. We record individual mutated runs as mutants rather than consider modified programs as mutants, and therefore the semantic mutation adequacy score can always be calculated. We also use random sampling and mutations weighted by likelihood, rather than considering every mutant equal. SMT uses data flow analysis of Java bytecode to avoid program crashes caused by mutating array indices outside array index range.

In chapter 6, we ran experiments to compare MT against SMT, and found SMT, in both white-box and black-box variants, to be better at comparing quality of verifiers (DBC contracts). We also observed that a number of shortcomings of traditional MT do not apply to SMT. Most significantly:

- SMT produces mutants that have small behavioral difference compared to original program.
- SMT mutation adequacy score can actually be calculated, and can be calculated without human intervention.
- SMT adequacy score uses the full range of 0.0 - 1.0
- SMT is much more likely to give strictly higher scores to more strict verifiers that pass the original program.

Our black-box mutation operators were richer than our white-box mutation operators in these experiments, and black-box SMT (SMTb) performed better, and actually got a perfect score on desired features:

- SMTb gave 0.0 only to empty contract, and 1.0 only to complete contract
- SMTb always gave strictly higher scores to strictly better verifiers

As compared with SMTb, white-box SMT (SMTw) did not have any array-size changing mutation operation, and therefore failed to distinguish any two verifiers that differ only in measuring array length. Other than two such cases, SMTw also gave strictly better scores to strictly better verifiers. Traditional MT failed to give empty contract 0.0, and failed to give strictly better score in 6 out of 11 cases of strict subsumption.

SMT has been designed to be integrated within VERDICTS, to measure and compare verifiers and assign adequacy scores so that the user can be sure that verifier evolution better represents the current behavior of the component being tested. We believe this quality feedback can greatly speed discovery of specifications, in the form of contracts that can also be used as automated verifiers. We believe that this interaction would also improve both the quality and the efficiency of software analysis/comprehension and testing.

Appendices

Appendix A

Software Comprehension Theories

A.1 Direction of Comprehension, Opportunistic Switching Strategy

Earlier theories of program understanding from 1970s and 1980s can be classified by direction of comprehension, into:

- Top-down models such as those by Brooks [20], by Soloway & Ehrlich's [75]),
- Bottom-up models like Pennington's model [69],
- Combined/opportunistic models that switch between these two strategies, such as Shneiderman's model [74] and Letovsky's model [57].

A.2 Von Mayrhauser & Vans' Integrated Meta-Model

Von Mayrhauser and Vans' well-researched integrated meta-model [59] combines elements of various top-down and bottom-up models and the concept of opportunistically switching between top-down and bottom-up processes. Studies based on observations often reveal a mixture of top-down and bottom-up activity as well in both procedural and object-oriented programming tasks performed by expert developers [59] [25].

Appendix B

Software Comprehension Issues

B.1 Software Comprehension is Vital, Yet Rarely Studied and Never Measured

Comprehension of software is vital in all software development, maintenance and reverse engineering tasks. Still, there is a general avoidance of studying software comprehension by software engineers, which may be in part due to software developers' distrust of theories and conclusions of "soft sciences".

Recall from chapter 1 the example of PCODA 2010 (International Workshop on Program Comprehension Through Dynamic Analysis), where no paper claims or tries to prove any improvement in comprehension. Even though the automatic data processing aspect is easier to evaluate, it is rather the soft science of actual software comprehension by individuals that we need to understand better.

Industry also ignores software comprehension. New developers have to catch up, and existing developers may often have to continually refresh their understanding of the system they are working on, to keep their mental models of the system aligned with the evolving system. Still, there is practically no practice of even attempting to measure a developer's level of comprehension of a particular piece of software.

Interested parties often presume that developers generally or completely comprehend the inner workings of software, and consider it offensive to ask to measure any developer's level of comprehension during software development as it questions developer competency.

Evaluating a software engineer is understandably a sensitive issue. But even though code ownership has also been a similar sensitive topic, industry has picked up the practice of code reviews which pierce through code ownership in open public debate of peers, to discover shortcomings of a piece of code that is often written by one developer.

Code reviews are often conducted carefully to avoid using results to judge any individual developer's level of comprehension or competency. There's often a promise of personal immunity so that question-answer sessions are not perceived as attack-defense, and criticisms about code are not later used to judge individual developers.

Code reviews publicly evaluate quality of code, with the goal of improving quality of software. There's no similar public or private evaluation of developers to judge level of software comprehension, with the ultimate goal of improving software quality.

B.2 Essential Incompleteness of Software Comprehension

B.2.1 Lines of Code, Years of Reading!

KLOC, kilo-lines-of-code, is a relatively simple measure of program complexity that has repeatedly been shown to be predictive of various measures of interest to software developers. How feasible is it to ask a developer to know everything about the software they are working on? As source code is the final word on all behavior under all conditions, the developer would have had to read all of source code to know everything about the software they are working on. How does this reading compare to other types of reading, such as a book?

One way to compare the size of the task is to just compare the number of lines of text that must be read. Here are some approximate numbers for one small and one large book, calculated by counting the lines on a few sample pages and multiplying out with the number of pages (using sample pages and information online):

- Animal Farm - 144 pp, about 5000 lines total; 5 KL (kilo-lines)
- Da Vinci Code - 454 pp, about 18000 lines total; 18 KL

Compare this with the statistics from 2001 for some open-source software:

- Apache: 80 KLOC (kilo-lines of code)
- Linux: 800 KLOC
- Mozilla: 2100 KLOC (2.1 million lines)

Humans average about 240 words per minute on text such as a book. At about 12 words per line, this corresponds to 20 lines per minute. At this rate, our books would take:

- Animal Farm: 250 minutes (4 hours)
- Da Vinci Code: 900 minutes (15 hours)

A book is designed to be consumed as a linear text whereas software is never implemented with that intention, and source code often consists of directories/packages/modules, and structured as a tree rather than a single linear text. What is more, nodes in the tree have cross-dependencies, and it is best to study software components as a highly interconnected graph, and such graph may even contain cyclic dependencies and strong components [1] [31].

Let us assume the speed is same as what was observed in a study of comprehension of a COBOL program [32], about 188 lines/hour, 1200 lines/day (assuming 6.4 hours/day). Then the software mentioned above would take:

- Apache: 67 work days $\approx$ 13 work weeks $\approx$ 3 months

[1] Whereever there are cyclic dependencies in a graph, there are sets of nodes called strong components where every node depends directly or indirectly on every other node in the set.

- Linux: 667 work days $\approx$ 133 work weeks $\approx$ 2.5 years
- Mozilla: 1750 work days $\approx$ 350 work weeks $\approx$ 7 years

Even though these are unmanageably impractical numbers for start-up time for a new developer on a project, these estimates are lower than what we should expect, as there are limits to how much we can remember, and in fact, if reading takes longer than a few weeks, much of what is read may be forgotten.

B.2.2 Prior Knowledge and Expertise

Beyond the impracticality of reading and knowing application source code in full, there are similar issues around expected prior knowledge and expertise for a developer.

Software development tasks and stages presume having a competent level of comprehension of the behavior/domain, programming context, and software-specific information. Beyond the software architecture, design, and implementation, the developer is expected to also know:

- Behavior & Domain: Required behavior of the program, domain/vertical knowledge.
- Context/Environment: Programming languages and libraries used, any executables that this software will interact with, and possibly, underlying operating system and hardware behavior and limitations.

A misunderstanding of any of these can cause reality to be misrepresented in the developer's mental models, and can cause software to contain a bug. Software verification by a second party, and later, validation by end user acceptance testing is vital because our mental models may be incomplete or faulty.

Today, most of this context also contains relatively large software whose behavior cannot be fully known without studying the source code, as documentation and specifications may not be correct or up to date. Therefore, similarly, for any nontrivial behavior, domain and context, prior knowledge cannot be complete either.

B.2.3 Simple Top-Down or Bottom-Up Traversal Continually Leaves Some Comprehension Questions Unanswered

There are actually various reasons why a part of program cannot be sufficiently comprehended in full without examining both top-down and bottom-up connections.

Comprehension proceeds with the “what”, “how” and “why” questions that developers ask themselves and try to answer while performing comprehension tasks [ref proposal]. Some questions, especially “how” a function works, can only be answered by examining context below the function, looking at functions called. These lower functions make up the smaller branches of the call hierarchy below current function. Other questions, especially “why” a function exists at all, can only be answered by looking at callers that are higher up in the call hierarchy. Some questions such as “what does this function do?” can best be answered by looking both up and down to understand why and how.

Therefore, no function can be understood completely in isolation; partial comprehension and hypotheses about other functions surround and sustain comprehension of any one function.

B.2.4 Bottom-Up Static Analysis and Top-Down Dynamic Analysis

Earlier “program understanding” models were restricted to static analysis, reading the source code, rather than executing the program. When observations from dynamic analysis are added (and especially when it can quickly be performed for modified program), the program behavior observed helps form high-level hypotheses while source code helps form low-level hypotheses, often without interconnecting links. Therefore, both top-down strategies from behavior/domain knowledge and bottom-up strategies from source code/programming language knowledge are useful and are often employed in comprehension tasks.

Developers regularly switch between top-down and bottom-up orientations for analysis and comprehension [59]. OO (OOP) comprehension has been observed to use more top-down than procedural programming, and both OOP and procedural programming are more likely to use more top-down (analysis and comprehension) early on and more bottom-up later, but generally employ both orientations throughout [25].

Conclusions:

- As recall is not 100%, it makes sense to not just traverse top-down or bottom-up exclusively, but rather go back and forth to refresh memory.
- Going only top-down, questions of implementation (operational details) will remain unanswered at each step, whereas going only bottom-up, the question of purpose and need will remain unanswered at each step.
- Comprehension cannot be performed by traversing one component at a time, with the goal of complete comprehension before analyzing related components.
- Partial specifications of related components must sometimes be sufficient in verifying complete specification for a component (otherwise comprehension could never converge for any component).
- As comprehension must often be partial, for task-oriented comprehension (say, of one component), it may be best to focus on the one component and its neighbors (up and down), and start with partial, evolving mental models and discovered specifications.

Appendix C

From Hoare Triple to DBC Contracts

Floyd-Hoare logic proposed by Hoare in 1969 [38] applies earlier work done by Floyd on flow charts [33] to create a formal system that can be used to prove the correctness of computer programs.

To formally declare the effect of executing a statement S in an imperative language, we need to define how this changes the program state. We can use two predicate logic assertions P and Q (called precondition and postcondition) to write the "Hoare triple":

```
{P} S {Q}
```

This Hoare triple states that, if precondition P holds before statement S executes, postcondition Q will hold after S has executed. P and Q can refer to any variable in scope of S.

Such a triple can often fully define the operation of a simple statement in an imperative programming language. Using Floyd-Hoare logic inference rules, we can discover Hoare triples for compound statements built out of simple statements and looping and block constructs. In the case of loops, it is often helpful to discover a loop invariant that is preserved by execution of the loop body:

```
{I} B {I}
```

Here, B is the statement block that makes up the loop body, and I is the invariant.

To extend this idea to procedural languages, we only need to note that statement S could just as well represent a procedure, and a function could be analyzed by storing the result in a meta-variable, in order to represent behavior of a function F formally:

```
{P} result = F {Q(result)}
```

We assume that result is declared to be of the same type as the return value of function F. In this case, postcondition Q can state requirements for result as well as any other side effects. Often, the same meta-variable name (such as "result", here) is used for any function, and assignment for result and Q's dependence on result are not explicitly stated. We simply write:

```
{P} F {Q}
```

Here, Q can refer to result but P cannot. Both P and Q can refer to any arguments this function accepts, by using the argument names in the function signature. Using the OCL syntax that we will soon see, the same DBC specifications look like:

```
pre: P
method F
post: Q
```

To extend this idea further to object-oriented languages, we need to add the idea of class invariant (also called object invariant), which states the internal state consistency requirements for the object. Obviously this only applies when there is a "this" object. When an invariant-preserving method is called, it may temporarily allow invariants not to hold during its execution, but must clean everything up before termination so that all invariants hold. Usually, there are private helper methods that the class's public methods can use even when class invariants do not hold. For this reason, invariants are often considered preserved only by the public instance methods of the class; private/protected and static methods are not required to preserve the invariants. For a method M that must preserve the class invariant, the combined requirement can be written as a Hoare triple as:

```
{I & P} M {I & Q}
```

This states that Method M will preserve class invariant I while satisfying postcondition Q, if invariant I and precondition P are satisfied before the method call. Class invariants should be preserved by all public instance methods, so class invariant declaration is usually associated with the class rather than its methods.

In OOP, all predicates (invariants, pre- and post-conditions) can:

- refer to object fields,
- use side-effect-free methods of any class, and
- create and manipulate contract-local variables and objects.

Beyond this, preconditions and postconditions can access method interface:

- Preconditions and postconditions can refer to method arguments.
- Postconditions can refer to result.
- Postconditions can also refer to state of a variable (or possibly any expression) before the method call, as we will see next.

To be able to compare state before and after method invocation easily, postcondition predicates may need to refer to older value of some variable x (argument or field) from before the method invocation, often using a syntax such as x' or x@pre. In a Hoare triple, this syntax would allow us to write:

```
{} x++; {x == x@pre + 1}
```

By using contract-local variables, same feat could be achieved for an int x with:

```
{int xpre = x} x++; {x == xpre + 1}
```

Even though we used a single-operation statement here instead of a method that increments a variable's value, it is clear how the same idea applies to any method with possibly modified values. These two contracts for plusOneLetterGrade() method in Grade class are equivalent (and use the object field "double grade" in Grade class, so grade := this.grade):

```
{} Grade.plusOneLetterGrade() { grade = grade@pre + 1.0; }
{ double gradepre = grade; }
    Grade.plusOneLetterGrade() { grade = gradepre + 1.0; }
```

There is a change in focus as we move from Hoare triples to DBC contracts. Floyd-Hoare logic:

- Focuses on individual statements
- Is designed to prove programs correct, by analyzing all statements
- Is a white-box approach that must have access to all code

In contrast, DBC:

- Focuses on individual methods
- Is designed to post monitors/gatekeepers/verifiers for inter-method communication, at method entry and exit points.
- Barring recursion, is not a white-box approach, and can be applied to a method without knowing its code; method may even be called by remote invocation, and may execute in another system.
- Cannot formally prove a method implementation to be correct
- Is instead used as a monitoring mechanism during component testing and integration testing.

Method focus makes DBC much more lightweight and efficient compared to analyzing and formally proving every statement correct.

Appendix D

DBC Tools, Languages

D.1 DBC Tools, Languages

This section briefly compares three DBC languages before stating why we prefer to use the Object Constraint Language (OCL) syntax:

- Eiffel: Meyer's own language that is designed to be DBC-capable
- OCL: Object Constraint Language, part of UML standard, used in UML static class diagrams and in model-driven engineering.
- JML: Java Modeling Language; has more advanced features and the concept of model objects that are only used to verify contracts. Also defines a large set of side-effect-free collections classes for Java, in the style of functional programming.

In Meyer's Eiffel language with native DBC support, the syntax to declare a method uses "require/ensure" pair to declare preconditions and postconditions:

```
method(arg:Type) is
require
  P   -- precondition
do
```

```
    ... -- method implementation goes here
  ensure
    Q   -- postcondition
  end
```

In OCL, designed to accompany UML diagrams to help specify behavior through constraints, the preconditions and postconditions are listed with "pre:" and "post:", after specifying the method context:

```
context Class::method(arg:type)
pre: P
post: Q
```

OCL syntax for class invariants specifies class context:

```
context Class
inv: I
```

In almost any language used to declare DBC contracts, multiple conditions can be listed one by one. This is equivalent to AND-ing these conditions. This:

```
pre: P1
pre: P2
```

is equivalent to:

```
pre: P1 and P2
```

Most languages also allow for handling exception cases, which define what types of invalid inputs would cause which types of exceptions to be thrown.

Java Modeling Languge (JML) borrows algebraic specification language ideas from Larch, and allows keeping model objects that are created and maintained through the life cycle of a modeled object. Such objects are created only if DBC verification is turned on These objects can only be accessed in JML specifications; they are not visible to any method's implementation. For many everyday classes with side effects, the states and operations are often modeled in JML through compositions of objects that instantiate JML's side-effect-free classes. These side-effect-free classes ("algebras" in Larch) can only create immutable objects, in effect allowing functional OO programming. For example, Set.remove(Element) creates another immutable Set, with the element removed. Immutability allows sharing collections to create other collections, which allows for reasonably efficient implementation that does not require incessant cloning.

JML also allows for multiple specification cases to be separately declared with preconditions and postconditions, such as:

```
  requires sign > 0
  ensures result == arg
also
  requires sign < 0
  ensures result == -arg
also
  requires sign == 0
  signals_only ArithmeticException
```

Ignoring the error condition with sign == 0, the other two cases could be combined into a single case in other languages such as OCL: (if sign value may change, replace sign with sign@pre everywhere)

```
pre: sign != 0
post: (sign > 0  &&  result = arg) || (sign < 0  &&  result = -arg)
```

In JML, declarations are embedded inline within Java code. All JML declarations are wrapped in "/@ ... @/", resembling, but different in syntax from multi-line comments (the lined-up vertical column of '@' symbols are not required below):

```
/@
 @  requires sign > 0
...
 @  signals_only ArithmeticException
 @/
public int negate(int arg, int sign) throws ArithmeticException {
  // method declaration goes here
  ...
}
```

As DBC expressions must be side-effect-free, '=' is used in OCL as equality comparison, rather than assignment. A local variable can be declared, assigned, and used in an OCL expression (with the syntax "let a = ... in <contract>"), but its value cannot change, and non-local external entities cannot be assigned new values.

DBC tools also have to declare which methods in the class are side-effect-free, and therefore could be safely called from DBC contracts. Some commonly used side-effect-free methods in Java classes are:

- getters: get... () methods, length() and size() methods
- substructure access: such as charAt(int index)
- query methods: isEmpty(), endsWith(String str), ...

OCL calls all side-effect-free methods "query" methods, and requires them to be marked with "query" to declare them safe for use in OCL expressions. JML calls these methods "pure" methods, and also requires that they be declared, with "pure":

```
/*@ pure @*/
public char charAt(int index) {
   return chars[index];
}
```

We mostly use OCL's "pre:" and "post:" for reasons of brevity, but we often skip the context declaration. Often, the contract is a general example that applies to multiple classes and methods (multiple contexts). At other times, the context is obvious from the preceding text. Even though we use OCL's tags, we do not use OCL's types, but rather stay with standard Java types (int instead of Integer, double instead of Real, etc).

D.2 What a Method Delivers: Postcondition Minus Precondition

The contract for a method is:

- IF the caller satisfies preconditions before call,
- THEN, the method will satisfy postconditions upon termination.

In essence, the work promised by the method through a contract is in the difference between postcondition Q and precondition P; it is in what Q promises above and beyond P. An empty method would satisfy any contract with $P \equiv Q$:

```
pre: X
post: X
```

For both P and Q, the least restrictive condition is "true", which is readily satisfied by any program state, and the most restrictive condition is "false", which cannot be satisfied by any program state. The "true" case never needs to be stated, as it adds nothing. This contract is equivalent to empty/missing contract (no precondition or postcondition) as it does not really require or promise to deliver anything:

```
pre: true
post: true
```

Beyond these two trivial cases, appendix E examines various types and patterns of contracts, noting some degenerate cases as well as some commonly useful patterns.

D.3 DBC Examples

Let us look at some examples of contracts. Consider the four methods shown in table D.1.

Table D.1: Four sample methods

Method	Signature	Description
int sqrt	(int n)	finds integer (truncated) squareroot
int findNth	(int[] ar, int n)	finds (n+1)-th highest value in array (n is 0-relative; n=0 finds max value)
int div	(int num, int denom)	performs integer division
int findInd	(double[] sAr, double d)	finds index of d in sAr, a sorted array (d must be in sAr; sAr is sorted up)

The corresponding preconditions and postconditions for these methods can be written in Java as seen in table D.2.

Table D.2: Preconditions and postconditions for four methods written as Java boolean expressions using helper functions "sortDesc", "isSorted" and "asSet".

Method	Precondition	Postcondition
sqrt	n >= 0	result == (int)Math.sqrt(n)
findNth	0 <= n && n < ar.length	result == sortDesc(ar)[n]
div	denom != 0	result == num / denom
findInd	isSorted(sAr) && asSet(sAr).contains(d)	sAr[result] == d

In defining these, we presumed some helper functions to have been defined:

- boolean isSorted(double[] ar): true if ar is sorted up (in nondescending order); ar[i+1] >= ar[i].
- int[] sortDesc(int[] ar): Returns a permutation of ar with elements sorted down (in non-ascending order); result[i+1] <= result[i].
- Set<double> asSet(double[] ar): Returns a set that contains all elements in ar.

Note that it is not as important for specifications to be efficiently computable [1]; it is often more preferable to use simple, short, easy to read specifications. The best time complexity for findNth method above (to find the n-th element in array ar) is O(log m) where m is the size of the array. But the postcondition for findNth refers to a sorted version of the array, which would minimally take O(m * log m) time. Time complexity of postconditions is not very important as they are often turned off after component testing.

Let us see how we can use OCL syntax to state these same preconditions and postconditions. In OCL, the method signatures are defined as:

```
sqrt(n:Integer):Integer

findNth(ar:Integer[], n:Integer): Integer

div(num:Integer, denom:Integer): Integer

findInd(sAr:Real[], d:Real): Integer
```

Table D.3 shows the preconditions and postconditions in OCL syntax.

Table D.3: Preconditions and postconditions for four methods written in OCL syntax

Method	Precondition	Postcondition
sqrt	n >= 0	(result*result <= n) and (n < (result+1)*(result+1)))
findNth	0 <= n and n < ar.length	let nGrtr: Integer = ar->select(x | x>result)->size in (n >= nGrtr) and (n < nGrtr + ar->count(result))
div	denom <> 0	result = num / denom

findInd	sAr->count(d)>0 and
	for i:Integer in 2..sAr->size |
	sAr[i-1] <= sAr[i];
		sAr[result] = d

[1] There are still exceptional cases where inefficiency can make component testing impractical, and should be addressed. Consider, for example, a class invariant that requires all objects of some type C2 to be checked for some property. Invariants are evaluated often, and if there are too many objects of type C2, the time complexity penalty becomes too large to be ignored. As a practical example, in a college's student registration system, a StudentID class should not state as a class invariant that there is one and only one Student object that uses this StudentID.

Note that for findNth, we did not need to assert separately that result exists in the array (with ar->count(result) > 0), because if this result does not exist, the two statements about n and nGrtr become contradictory and unsatisfiable together ((n >= nGrtr) and (n < nGrtr), when ar->count(result) is 0).

One way to verify a Java program against an OCL specification is to convert OCL declarations to Java method verifiers. Starting from OCL, this can be done manually or by an OCL tool.

Even without the knowledge of the declarative OCL language, we can directly use the programmatic syntax of the Java language itself to create method verifiers. Table D.4 shows same methods and contracts as above, this time written in full in Java, without helper methods. For space efficiency, we use $r to represent result in this table.

Table D.4: Preconditions and postconditions for four methods written as Java boolean expressions using only Java instructions and assert statements

Method	Precondition	Postcondition
sqrt	assert n >= 0;	assert ($r*$r <= n) && (n < ($r+1)*($r+1)));

findNth	assert (0<=n) && (n<ar.length);
	// n is a 0-relative index
		int nGrtr = 0; int nEq = 0;
		for(int val: ar)
		if (val > $r) nGrtr++;
		else if (val == $r) nEq++;
		assert (n >= nGrtr)
		&& (n < nGrtr + nEq);

div	assert denom != 0	assert $r == num / denom;

findInd	for(int i=1; i<sAr.length; i++)
	assert sAr[i-1] <= sAr[i];
	int j = 0;
	while(j<sAr.length && sAr[j]!=d)
	j++;
	assert j<sAr.length; // sAr[j]==d
		assert sAr[$r] == d;

For simplicity, we have the somewhat circular definition of using integer division to define the postcondition of integer division. A better postcondition would just use multiplication to define integer division, handling different cases of signs of numerator and denominator as needed. For this div method, a human-readable but not automatically testable API documentation may look like this:

```
/** Divides n by m; m should not be 0. */
int div(int n, int m) { return n/m; }
```

Even though integer division is defined for negative values of n and m as well, it is easier to describe the behavior for positive n and m. A partial contract for div may only allow operation with nonnegative n and positive m values. Here, we mark preconditions with @pre and postconditions with @post:

```
pre :  n >= 0  &&  m > 0
post:  result * m >= n
post:  (result - 1) * m < n
```

Integer division satisfies this contract, but this contract does not allow negative values of n and m. Integer division can operate in conditions not allowed by this contract; it works on all four quadrants, not only one.

A more general complete contract can easily be derived by requiring these conditions on absolute values of n and m (leaving only m != 0 as the precondition), as well as requiring consistency of sign in result whenever result is not 0. Using traditional absolute value function 'abs', and a signum function 'sign' that returns -1 for negative integers, +1 for positive values, and 0 for 0 (actually, using +1 or -1 for 0 also works for our contract):

```
pre :  m != 0
post:  result == 0 || (sign(result) == sign(n) * sign(m))
post:  abs(result) * abs(m) >= abs(n)
post:  (abs(result) - 1) * abs(m) < abs(n)
```

The sign consistency line states that result is negative if n or m is negative, but not when both n and m are negative. This line could also be written using simpler binary valued comparison operators and boolean value comparison instead of ternary signum operator and multiplication:

```
post:  result == 0 || ((result < 0) == ((n < 0) != (m < 0)))
```

Alternatively, we can use the logical XOR operator, `^`:

```
post:  result == 0 || ((result < 0) == ((n < 0) ^ (m < 0)))
```

Appendix E

Types of DBC Contracts, and How to Read Them

E.1 Degenerate Contracts and Extreme Cases

Let us first consider some degenerate and extreme contracts. Here's the weakest possible contract:

```
pre: false
  (This can be combined with any postcondition Q, even "post: false"!
   Method may crash every time, and will still pass this contract!)
```

This contract does not require the method to do nothing. The method may even crash every time it is called, and yet it still trivially satisfies this contract.

Next weakest possible contract requires normal termination, but not anything else from a method:

```
post: true
  (When combined with any precondition except false:
   Method must terminate, but promises nothing else)
```

The following method contract promises to deliver nothing except for not destroying a precondition I (which is in essence an invariant):

```
pre: I
post: I
  (Method only promises to preserve I;
   empty method body would satisfy this)
```

The following method contract promises to deliver less than nothing (it requires more than it delivers); it suggests that it may actually do some harm:

```
pre: I and P
post: I
  (Method must preserve I, but may cause ! P;
   it need not deliver anything else)
```

E.2 Categories of Everyday Contracts

Now we're ready to consider some categories of reasonable contracts we are likely to come across in DBC. These may appear syntactically similar to degenerate cases, but are actually categories of practical contracts we may use everyday in DBC.

This method contract promises to deliver Q without affecting I:

```
pre: I
post: I and Q
  (method must deliver Q while preserving I)
```

If I is merely the object invariant, it is already implicitly considered to be part of both precondition and postcondition, and need not be spelled out. In this case, we merely use:

```
post: Q
  (method must deliver Q while preserving object invariants)
```

The explicit repetition of I in both precondition and postcondition is only needed for method-specific invariants (that are required and preserved by this method) beyond the implicit object invariant (that are required and preserved by all public methods).

This method contract promises Q while destroying (consuming) P:

```
pre: P
post: Q
  (method must deliver Q, may cause ! P)
```

In this case, we are interested in anything that does not already fall under any of the simpler categories seen above (neither P ==> Q nor Q ==> P).

This type of contract is often used when delivering Q would naturally cause ! P because P and Q cannot be satisfied together:

```
Q ==> ! P
! (P and Q)
```

Appendix F

ASM Library for Java Bytecode Manipulation

What follows in the next few sections is a brief introduction to ASM. For a quick introduction that goes into a little more detail, we recommend ASM tutorial at ASM site [51]. For more examples and topics, please refer to the ASM Guide [21], which is very readable, but a little long at 138 pages.

F.1 Event-Based Sequential Access Parser (Similar to SAX for XML)

This type of parsing, code generation and manipulation is defined in ASM's Core package. It uses the Visitor design pattern's push approach; visitXyz methods you define (override from empty definition) for structural elements of Java bytecode will be called as ASM comes across each element.

A ClassReader can be used to read a class file (bytecode). A ClassVisitor can be associated with the ClassReader so that ClassReader will call ClassVisitor for whole and parts of the structure. There are four Visitor classes and various visit methods that can be used to traverse various structures that make up a class:

- ClassVisitor: `visit{, Source, OuterClass, Attribute, Annotation, EndField, Method, InnerClass, End}()`
- AnnotationVisitor: `visit{Xyz, End}()`
- FieldVisitor: `visit{Attribute, Annotation, End}()`
- MethodVisitor: `visit{AnnotationDefault, Attribute, Annotation, Parameter-Annotation, Code, Xyz, End}()`

Note: Xyz above is a place holder for various items that can be visited with the corresponding visit... methods in the visitor. For example, for visiting statements in a method, we have local variable access instruction, field access instruction, load constant instruction, jump instruction, label (for jump), ..., which can be visited with the `visit{VarInsn, FieldInsn, LdcInsn, JumpInsn, Label, ...}()` methods.

Code that visits every part of a class may look like the following:

```
ClassVisitor cv = ...
cv.visit(): Informs visitor of top-level class declaration information:
         version, access rights, name of class, superclass name, interfaces
cv.visitSource();
cv.visitOuterClass();
for each class attribute:
  cv.visitAttribute();
for each class annotation:
  AnnotationVisitor av;
    = cv.visitAnnotation(description, isVisible);
  for each part of the annotation:
    av.visitXyz(); // Xyz: various parts of annotations
  av.visitEnd()
for each field:
  FieldVisitor fv
    = cv.visitField(access, name, description, signature, value);
  for each field attribute:
    fv.visitAttribute();
  for each field annotation:
    fv.visitAnnotation();
  fv.visitEnd();
for each method:
  MethodVisitor mv
    = cv.visitMethod(access, name, description, signature, exceptions);
  mv.visitAnnotationDefault();
  for each method attribute:
    mv.visitAttribute();
  for each method annotation:
    mv.visitAnnotation();
  for each method parameter annotation:
    mv.visitParameterAnnotation();
  mv.visitCode();
  for each statement in method body:
    mv.visitXyz();  // Xyz: to visit various code structures
  mv.visitEnd();
for each inner class:
  cv.visitInnerClass(name, outername, innername, access);
cv.visitEnd();
```

Here, the visitor is a bytecode observer or consumer. Multiple visitors can delegate part or whole task to a next level visitor, by using the Chain of Responsibility pattern, using visitors derived from ClassAdapter and MethodAdapter.

This allows creation of a chain with a ClassWriter at the end of the chain. For example, if we have created an InstrumentingVisitor and a MutatingVisitor that are derived from ClassAdapter, we can read, mutate class, instrument it, and write it to a class file, by creating the responsibility chain from the end backwards like this:

```
// read our class bytes (bytecode) for input
byte[] inbytes = ...;
// this will hold the bytecode for transformed class
byte[] outbytes;


// Desired chain:
//    read(inbytes[]) -> mutate() -> instrument()
//    -> write(outbytes[])
ClassWriter cw = new ClassWriter();
// instrument -> write   link of the chain
ClassVisitor ci = new InstrumentingVisitor(cw);
// mutate -> instrument  link of the chain
ClassVisitor cm = new MutatingVisitor(ci);
// read
ClassReader cr = new ClassReader(inbytes);
// Chain is ready, processing can start now;
// ClassReader ->   (accept as visitor)
//    MutatingVisitor -> InstrumentingVisitor -> ClassWriter
cr.accept(cm, 0);
// convert bytecode from ASM structures to byte[]
outbytes = cw.toByteArray();
```

For efficiency purposes, ASM actually prefers if you let ClassWriter know the ClassReader, so that in cases of unmodified method bytecode, writer can just use reader's bytecode for method. This optimization can be used by declaring the reader first and passing its reference to the writer (other declarations remain the same):

```
...
ClassReader cr = new ClassReader(inbytes);
ClassWriter cw = new ClassWriter(cr, 0);
...
```

Instead of supplying the bytes, one can also supply an input stream as in:

```
InputStream in = new FileInputStream("Mutater.class");
ClassReader cr = new ClassReader(in);
...
```

Just like SAX parser for XML, the event-based Java bytecode manipulation that the ASM Core package has speed and memory advantages, but cannot be used as easily for some types of analysis and transformations. Compared to the tree-based (DOM-like) parser in the next section, ASM Core package, used as-is (without creating a memory-resident parse tree), with a single pass:

- is faster
- has smaller memory footprint (as document parse tree is never held in the memory)
- is less powerful; allows fewer types of analyses and transformations
  - can only be used for order-preserving manipulations of the bytecode
  - can only be used if knowledge gained from a later portion of the bytecode (for example, "is this private method ever called in this class?") need not be used to transform an earlier portion of the bytecode.

In the next section, we will see ASM's Tree API that creates a memory-resident parse tree structure which allows more complicated types of analysis and transformations. Similar to common practice with DOM parsers for XML built on top of SAX parsers, ASM's Tree Parser is built on top of efficient event-based parser that we saw in this section.

F.2 ASM's Tree API

As with SAX/DOM distinction for XML document parsing and manipulation, as compared with ASM Core API that provides an event-based sequential access parser, ASM's Tree API:

- Uses more memory to hold the whole class file parse tree in memory.
- Keeps substructures in a form that is easy to access, analyze and manipulate.
- Allows repeated access to substructures in any order.
- Therefore accomodates many types of structural manipulations that are much harder to perform with ASM Core API.

In our projects, we initially used the event-based manipulation, but had to move later to Tree API as we needed more complex analysis and manipulation of the Java classes.

To generate new structures in ASM's Tree API, we can create new nodes that correspond to various structural elements of the Java bytecode (and therefore, of Java class), such as:

```
ClassNode, FieldNode, MethodNode
XyzInsnNode: {Abstract, Field, Ldc, Method, Var, Jump, Type, }InsnNode
InsnList
```

The last one, InsnList holds all instructions of a method. All the instruction nodes are subclasses of AbstractInsnNode, and each instruction node object can only belong to one InsnList (where it can appear only once), in one Method. To copy an instruction, we have to clone/copy the object. We can also move an instruction (remove from first list, add to the second).

The Tree API is built on top of the event-based parser API. In fact, ClassNode is a ClassVisitor. To read a class into memory to create the parse tree for the class, we only need to create an empty ClassNode and let it visit a ClassReader, as in:

```
ClassReader cr = new ClassReader(inbytes);
ClassNode cls = new ClassNode();
cr.accept(cls, 0);
```

F.3 Data Flow Analysis with ASM

As part of ASM's tree API, the `org.objectweb.asm.tree.analysis` package provides functionality to help perform static code analysis on the memory resident parse tree of the Java class, for:

- data flow analysis (forward and backward analysis)
- control flow analysis

We focus below on data flow analysis. Please refer to ASM guide and Javadocs for more information on control flow analysis, which can, for example, be used for cyclomatic complexity analysis.

As JVM is a stack-based machine, data flow analysis has to use a stack, with proper positions in the stack accessed by each instruction that uses the stack. ASM's data flow analysis basically performs a symbolic execution of all the instructions of the method, and uses "Value" objects to represent all possible values:

- for any field, method argument, method return value, and local variable

- at any place in the stack during any bytecode instruction (note: stack is empty between Java statements, but may hold data between JVM instructions that collectively make up each Java statement)

Instead of single values, these Value objects are data structures that represent all possible values at a given point in the code. This is a major difference between JVM and data flow analysis: JVM would use individual value, for example one integer, for value of an integer variable. Data flow analysis instead may hold a set of integers that to list all possible values this integer variable may have at any given point in Java bytecode execution. Our analysis goals may not make distinctions between some values, and holding the set of all possible values at any given point in execution may become too complicated. Therefore, an important aspect of all possible values, rather than the set of all possible values, will usually be held in the data structures that represent values used in data flow analysis.

The "Value" objects used in data flow analysis cannot hold any actual integer, double or pointer values; for example, local variable i's value cannot be represented with just Integer(10), as this would mean:

- the variable is not varying, so it is not really a variable, it is a constant
- we have chosen to represent the single-element set by the single element itself instead of a uniform structure that would allow any number of values as elements; this requires a switch-case statement in any processing method.

As a simple example of data flow analysis, consider the task of discovering the minimum and maximum values of integer and double-valued variables and fields. A possible data structure that can be used to handle some basic arithmetic operations is:

```
public class IntRange    implements Value { int min, max;    }
public class DoubleRange implements Value { double min, max; }
```

Now, for any integer-valued variable, we can use an IntRange value to discover and use its minimum and maximum possible value. Note that this approach is relatively crude and may produce relatively large ranges. For example, if n can only have the values 0 and 1024, we have to record this as n having any value in the range 0 - 1024. Then, all we can say about m = n % 100 is that it is in range 0 - 99 rather than that it may only be 0 or 24.

Data flow analysis proceeds by taking all paths, executing each jump once.

The loops are not run multiple times (as variables do not have values, we would not know how many times to run the loops). The switch-case statements are executed by going to all possible cases. Both if and else branches run. Jump instructions to jump backwards in code can cause various other values to be in the stack at the jump-to location in the bytecode.

One way to mix forward and backward data flow analysis is to use ASM API for forward data flow analysis, with complex Value structures that record accumulative history of values, and do further analysis on these Value structures to track causal links backwards.

We have used ASM's data flow analysis classes with values that represent origin of value (a bytecode instruction that creates the value, an argument to method, a field, or a local variable). As we need to hold the union of all possible values, we use sets of origins of values that may be held at any point during the instruction. This allows us to go backwards and find responsible parties and modify code to limit their value variability, to make sure that the value used at a point during the instruction conforms to what is acceptable.

For more information about using ASM Tree API, please refer to ASM Guide and this article: Manipulating Java Class Files with ASM 4 - Part Two: Tree API.

Bibliography

[1] Pekka Abrahamsson et al. *Agile software development methods*. Tech. rep. VTT Publications, 2002.

[2] Hiralal Agrawal. "Towards automatic debugging of computer programs". UMI Order No. GAX92-01293. PhD thesis. West Lafayette, IN, USA, 1992.

[3] Hiralal Agrawal and Joseph Robert Horgan. "Dynamic Program Slicing". In: *Proc. ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. 1990, pp. 246–256.

[4] Hiralal Agrawal et al. *Design of Mutant Operators for the C Programming Language*. Tech. rep. SERC-TR41-P. West Lafayette, IN: Software Engineering Research Center, Purdue University, 1989.

[5] Francis J. Anscombe. "Graphs in statistical analysis". In: *American Statistician* (1973), pp. 17–21.

[6] *Aspect-Oriented Programming: Implementations*. Wikipedia, URL: `http://en.wikipedia.org/wiki/Aspect-oriented_programming#Implementations` (visited on 02/08/2013).

[7] *AspectJ Project*. The Eclipse Foundation, URL: `http://www.eclipse.org/aspectj/` (visited on 02/08/2013).

[8] *Azureus/Vuze: Free Communications software downloads at SourceForge.net*. SourceForge, URL: `http://sourceforge.net/projects/azureus/` (visited on 02/21/2013).

[9] J. Bach. *Exploratory Testing Explained.* Tech. rep. Satisfice, Inc., 2003. URL: `http://www.satisfice.com/articles/et-article.pdf`.

[10] Luciano Baresi and Michal Young. *Test Oracles.* Tech. rep. 2001.

[11] David W. Binkley and Keith B. Gallagher. "Program slicing". In: *Advances in Computers* 43 (1996), pp. 1–50.

[12] Paul E. Black, Vadim Okun, and Yaacov Yesha. "Mutation Operators for Specifications". In: *Proceedings of the 15th IEEE international conference on Automated software engineering.* ASE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 81–. ISBN: 0-7695-0710-7. URL: `http://dl.acm.org/citation.cfm?id=786768.786981`.

[13] S. Kanat Bolazar and James W. Fawcett. "Debugging with Software Visualization and Contract Discovery". In: *SEDE.* Ed. by Walter Dosch and William Perrizo. ISCA, 2006, pp. 47–50.

[14] S. Kanat Bolazar and James W. Fawcett. "Measuring Component Specification-Implementation Concordance with Semantic Mutation Testing". In: *CATA.* Ed. by Wei Li. ISCA, 2011, pp. 102–107. ISBN: 978-1-880843-80-2.

[15] S. Kanat Bolazar and James W. Fawcett. "VERDICTS : Visual Exploratory Requirements Discovery and Injection for Comprehension and Testing of Software". In: *Software Engineering Research and Practice.* Ed. by Hamid R. Arabnia et al. CSREA Press, 2010, pp. 233–239. ISBN: 1-60132-167-8.

[16] *Bolazar's Research Page.* Syracuse University, May 2013. URL: `http://www.ecs.syr.edu/faculty/fawcett/handouts/research/Bolazar/index.html`.

[17] Bob Boothe. "Efficient algorithms for bidirectional debugging". In: *PLDI.* Ed. by Monica S. Lam. ACM, 2000, pp. 299–310. ISBN: 1-58113-199-2.

[18] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. “From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing”. In: *In ISSTA ’06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ACM Press, 2006, pp. 169–180.

[19] Frederick P. Brooks Jr. *The mythical man-month (anniversary ed.)* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-83595-9.

[20] Ruven E. Brooks. “Towards a Theory of the Comprehension of Computer Programs”. In: *International Journal of Man-Machine Studies* 18.6 (1983), pp. 543–554.

[21] Eric Bruneton. *ASM 4.0: A Java bytecode engineering library*. Sept. 2011. URL: `http://download.forge.objectweb.org/asm/asm4-guide.pdf`.

[22] David W. Bustard and Adam C. Winstanley. “Making Changes to Formal Specifications: Requirements and an Example”. In: *IEEE Trans. Software Eng.* 20.8 (1994), pp. 562–568.

[23] Holger Cleve and Andreas Zeller. “Locating causes of program failures”. In: *ICSE*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. ACM, 2005, pp. 342–351.

[24] David Coppit and Jennifer M. Haddox-Schatz. “On the Use of Specification-Based Assertions as Test Oracles”. In: *SEW*. IEEE Computer Society, 2005, pp. 305–314. ISBN: 0-7695-2306-4.

[25] Cynthia L. Corritore and Susan Wiedenbeck. “An exploratory study of program comprehension strategies of procedural and object-oriented programmers”. In: *Int. J. Hum.-Comput. Stud.* 54.1 (2001), pp. 1–23.

[26] M. Csíkszentmihályi. *Finding Flow: The Psychology of Engagement With Everyday Life*. MasterMinds Series. BasicBooks, 1997. ISBN: 9780465024117. URL: `http://books.google.com/books?id=HBod-fUzmBcC`.

[27] Márcio E. Delamaro, José; C. Maldonado, and Aditya P. Mathur. “Interface Mutation: An Approach for Integration Testing”. In: *IEEE Transactions on Software Engineering* 27 (2001), pp. 228–247. ISSN: 0098-5589. DOI: `http://doi.ieeecomputersociety.org/10.1109/32.910859`.

[28] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer”. In: *Computer* 11.4 (Apr. 1978), pp. 34–41. ISSN: 0018-9162.

[29] Richard A. DeMillo and A. Jefferson Offutt. “Constraint-Based Automatic Test Data Generation”. In: *IEEE Trans. Softw. Eng.* 17.9 (Sept. 1991), pp. 900–910. ISSN: 0098-5589. DOI: `10.1109/32.92910`. URL: `http://dx.doi.org/10.1109/32.92910`.

[30] *Design By Contract: Language Support.* Wikipedia, URL: `http://en.wikipedia.org/wiki/Design_by_contract#Language_support` (visited on 02/08/2013).

[31] James W. Fawcett, Murat K. Gungor, and Arun V. Iyer. “Analyzing Static Structure of Large Software Systems”. In: *Software Engineering Research and Practice.* Ed. by Hamid R. Arabnia and Hassan Reza. CSREA Press, 2005, pp. 491–496. ISBN: 1-932415-50-5.

[32] P. Fiore, F. Lanubile, and G. Visaggio. “Effort Estimation for Program Comprehension”. In: *Proceedings of the 4th International Workshop on Program Comprehension (WPC ’96).* WPC ’96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 78–. ISBN: 0-8186-7283-8. URL: `http://dl.acm.org/citation.cfm?id=525394.837837`.

[33] R. W. Floyd. “Assigning Meaning to Programs”. In: *Proceedings of the Symposium on Applied Maths.* Vol. 19. AMS, 1967, pp. 19–32.

[34] John B. Goodenough and Susan L. Gerhart. “Toward a Theory of Test Data Selection”. In: *IEEE Trans. Software Eng.* 1.2 (1975), pp. 156–173.

[35] John B. Goodenough and Susan L. Gerhart. "Toward a Theory of Testing: Data Selection Criteria". In: *Current Trends in Programming Methodology* 2 (1977). Ed. by Raymond T. Yeh, pp. 44–79.

[36] B. J. M. Grun, D. Schuler, and A. Zeller. "The impact of equivalent mutants". In: *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on.* IEEE. 2009, pp. 192–199.

[37] Abdelwahab Hamou-Lhadj et al. "Workshop on Program Comprehension through Dynamic Analysis (PCODA10)." In: *WCRE*. Ed. by Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky. IEEE Computer Society, 2010, pp. 279–280.

[38] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: `10.1145/363235.363259`. URL: `http://doi.acm.org/10.1145/363235.363259`.

[39] Chanika Hobatr and Brian A. Malloy. "The design of an OCL query-based debugger for C++". In: *SAC*. ACM, 2001, pp. 658–662.

[40] Chanika Hobatr and Brian A. Malloy. "Using OCL-Queries for Debugging C++". In: *ICSE*. Ed. by Hausi A. Müller, Mary Jean Harrold, and Wilhelm Schäfer. IEEE Computer Society, 2001, pp. 839–840. ISBN: 0-7695-1050-7.

[41] *Instructions Per Second.* Wikipedia, URL: `http://en.wikipedia.org/wiki/Instructions_per_second` (visited on 02/21/2013).

[42] S. A. Irvine et al. "Jumble Java byte code to measure the effectiveness of unit tests". In: *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007.* IEEE. 2007, pp. 169–175.

[43] *Java Tutorials - Reflection API Trail.* Oracle, URL: `http://docs.oracle.com/javase/tutorial/reflect/index.html` (visited on 02/08/2013).

[44] Yue Jia and Mark Harman. "Higher Order Mutation Testing". In: *Journal of Information and Software Technology* 51.10 (2009), pp. 1379–1393.

[45] Mira Kajko-Mattsson, Stefan Forssander, and Gunnar Andersson. "Software problem reporting and resolution process at ABB Robotics AB: state of practice". In: *Journal of Software Maintenance: Research and Practice* 12.5 (2000), pp. 255–285. ISSN: 1096-908X. DOI: `10.1002/1096-908X(200009/10)12:5<255::AID-SMR216>3.0.CO;2-L`. URL: `http://dx.doi.org/10.1002/1096-908X(200009/10)12:5<255::AID-SMR216>3.0.CO;2-L`.

[46] Mira Kajko-Mattsson, Stefan Forssander, and Ulf Olsson. "Corrective maintenance maturity model (CM3): maintainer's education and training". In: *Proceedings of the 23rd International Conference on Software Engineering*. ICSE '01. Toronto, Ontario, Canada: IEEE Computer Society, 2001, pp. 610–619. ISBN: 0-7695-1050-7. URL: `http://dl.acm.org.libezproxy2.syr.edu/citation.cfm?id=381473.381543`.

[47] Gregor Kiczales et al. "Aspect-Oriented Programming". In: *ECOOP*. 1997, pp. 220–242.

[48] Andrew J. Ko and Brad A. Myers. "Debugging reinvented: asking and answering why and why not questions about program behavior". In: *Proceedings of the 30th international conference on Software engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 301–310. ISBN: 978-1-60558-079-1. DOI: `10.1145/1368088.1368130`. URL: `http://doi.acm.org/10.1145/1368088.1368130`.

[49] *Koders.com*. Black Duck Software, Inc., URL: `http://www.koders.com/` (visited on 02/08/2013).

[50] *Krugle.org*. Aragon Consulting Group, Inc., URL: `http://www.krugle.org/` (visited on 02/08/2013).

[51] Eugene Kuleshov. *Introduction to the ASM 2.0 Bytecode Framework*. Aug. 2005. URL: `http://asm.ow2.org/doc/tutorial-asm-2.0.html` (visited on 02/08/2013).

[52] C. Larman and V.R. Basili. "Iterative and Incremental Development: A Brief History". In: *IEEE Computer* 36.6 (2003), pp. 47–56.

[53] Gary T. Leavens and Albert L. Baker. "Enhancing the Pre- and Postcondition Technique for More Expressive Specifications." In: *World Congress on Formal Methods.* Ed. by Jeannette M. Wing, Jim Woodcock, and Jim Davies. Vol. 1709. Lecture Notes in Computer Science. Springer, 1999, pp. 1087–1106. ISBN: 3-540-66588-9. URL: `http://dblp.uni-trier.de/db/conf/fm/fm1999-2.html#LeavensB99`.

[54] Gary T. Leavens et al. "JML (poster session): notations and tools supporting detailed design in Java". In: *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum).* OOPSLA '00. Minneapolis, Minnesota, USA: ACM, 2000, pp. 105–106. ISBN: 1-58113-307-3. DOI: `10.1145/367845.367996`. URL: `http://doi.acm.org/10.1145/367845.367996`.

[55] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. "Dynamic Query-Based Debugging". In: *ECOOP*. Ed. by Rachid Guerraoui. Vol. 1628. Lecture Notes in Computer Science. Springer, 1999, pp. 135–160. ISBN: 3-540-66156-5.

[56] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. "Query-Based Debugging of Object-Oriented Programs". In: *OOPSLA*. Ed. by Mary E. S. Loomis, Toby Bloom, and A. Michael Berman. ACM, 1997, pp. 304–317. ISBN: 0-89791-908-4.

[57] Stanley Letovsky. "Cognitive processes in program comprehension". In: *Journal of Systems and Software* 7.4 (1987), pp. 325–339.

[58] Bil Lewis. "Debugging Backwards in Time". In: *CoRR* cs.SE/0310016 (2003).

[59] Anneliese von Mayrhauser and A. Marie Vans. "Program Comprehension During Software Maintenance and Evolution". In: *IEEE Computer* 28.8 (1995), pp. 44–55.

[60] B. Meyer. "On formalism in specifications". In: *IEEE software* 2.1 (1985), pp. 6–26.

[61] B. P. Miller, L. Fredriksen, and B. So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[62] Matthias M. Müller, Rainer Typke, and Oliver Hagner. "Two Controlled Experiments Concerning the Usefulness of Assertions as a Means for Programming". In: *ICSM*. IEEE Computer Society, 2002, pp. 84–92. ISBN: 0-7695-1819-2.

[63] A. S. Namin, J. H. Andrews, and D. J. Murdoch. "Sufficient mutation operators for measuring test effectiveness". In: *Proceedings of the 30th international conference on Software engineering*. ACM. 2008, pp. 351–360.

[64] Cornelius Ncube, Patricia Oberndorf, and Anatol W. Kark. "Opportunistic Software Systems Development: Making Systems from What's Available". In: *IEEE Softw.* 25.6 (Nov. 2008), pp. 38–41.

[65] A. Jefferson Offutt and Roland H. Untch. "Mutation 2000: Uniting the Orthogonal". In: *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*. San Jose, California, 2001, pp. 34–44.

[66] A. Jefferson Offutt and Jeffrey M. Voas. *Subsumption of Condition Coverage Techniques by Mutation Testing*. Tech. rep. ISSE-TR-96-01. 4400 University Drive MS 4A5, Fairfax, VA 22030-4444 USA: Department of Information and Software Engineering, George Mason University, 1996.

[67] C. Pacheco et al. "Feedback-directed random test generation". In: *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE. 2007, pp. 75–84.

[68] *PCODA 2010 Proceedings*. Software Evolution Research Lab (SWERL) at Delft University of Technology, Delft, The Netherlands, Oct. 2010. URL: `http://swerl.tudelft.nl/bin/view/PCODA/PCODA2010#Proceedings`.

[69] Nancy Pennington. "Stimulus structures and mental representations in expert comprehension of computer programs". In: *Cognitive Psychology* 19.3 (1987), pp. 295–341. URL: `http://dx.doi.org/10.1016/0010-0285(87)90007-7`.

[70] Kevin Poulsen. *Tracking the Blackout Bug: Buried in four million lines of C code.* The Register, URL: `http://www.theregister.co.uk/2004/04/08/blackout_bug_report/` (visited on 04/20/2013).

[71] *Preliminary Specifications: Programmed Data Processor Model Three (PDP-3), by Digital Equipment Corporation.* Gutenberg, URL: `http://www.gutenberg.org/files/29461/29461-h/29461-h.htm` (visited on 02/21/2013).

[72] Winston W. Royce. "Managing the development of large software systems: concepts and techniques". In: *Proc. IEEE WESTCON*. Reprinted in Proc. Int'l Conf. Software Engineering (ICSE) 1989, ACM Press, pp. 328-338. IEEE Press, 1970.

[73] Ehud Y. Shapiro. *Algorithmic Program DeBugging.* Cambridge, MA, USA: MIT Press, 1983. ISBN: 0262192187.

[74] Ben Shneiderman and Richard E. Mayer. "Syntactic/semantic interactions in programmer behavior: A model and experimental results." In: *International Journal of Parallel Programming* 8.3 (1979), pp. 219–238. URL: `http://dblp.uni-trier.de/db/journals/ijpp/ijpp8.html#ShneidermanM79`.

[75] Elliot Soloway and Kate Ehrlich. "Empirical Studies of Programming Knowledge". English. In: *IEEE Transactions on Software Engineering* 10.5 (1984). Copyright - Copyright Institute of Electrical and Electronics Engineers, Inc. (IEEE) Sep 1984; Last updated - 2011-07-20; CODEN - IESEDJ; DOI - 7091025; 49101; 17010; IESEDJ; ISO; 00257501; 84-36064, pp. 595–595. URL: `http://search.proquest.com/docview/195578283?accountid=14214`.

[76] *SourceForge: Most downloads over all time.* SourceForge, URL: `http://sourceforge.net/top` (visited on 02/21/2013).

[77] M. Sridharan and A. S. Namin. "Prioritizing Mutation Operators Based on Importance Sampling". In: *Proc. 21st Intl. Symp. of Software Reliability Engineering.* 2010, pp. 378–387.

[78] R. H. Untch, A. J. Offutt, and M. J. Harrold. “Mutation analysis using mutant schemata”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 18. 3. ACM. 1993, pp. 139–148.

[79] Iris Vessey. “Expertise in Debugging Computer Programs: A Process Analysis.” In: *International Journal of Man-Machine Studies* 23.5 (1985), pp. 459–494. URL: `http://dblp.uni-trier.de/db/journals/ijmms/ijmms23.html#Vessey85`.

[80] J. M. Voas. “PIE: A dynamic failure-based technique”. In: *Software Engineering, IEEE Transactions on* 18.8 (1992), pp. 717–727.

[81] *Vuze - Wikipedia*. Wikipedia, URL: `http://en.wikipedia.org/wiki/Vuze` (visited on 02/21/2013).

[82] Andreas Zeller. “Automated Debugging: Are We Close”. In: *IEEE Computer* 34.11 (2001), pp. 26–31.

[83] Andreas Zeller. “Isolating cause-effect chains from computer programs”. In: *SIGSOFT FSE*. 2002, pp. 1–10.

[84] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005. ISBN: 1558608664.

[85] Andreas Zeller and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input”. In: *IEEE Trans. Software Eng.* 28.2 (2002), pp. 183–200.

VITA

NAME OF AUTHOR: Şefik Kanat Bolazar

PLACE OF BIRTH: Ankara, Turkey

DATE OF BIRTH: 17 July 1969

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

Syracuse University, Syracuse, New York

Middle Eastern Technical University, Ankara, Turkey

DEGREES AWARDED:

M.S. in Computer and Information Science, 1993, Syracuse University

B.S. in Computer Engineering, 1990, Middle Eastern Technical University

PROFESSIONAL EXPERIENCE:

Control Systems Programmer, Landis & Gyr, Zug, Switzerland, 1988

Manager, Software Development, Cagdas Bilgisayar (Computer),
Ankara, Turkey, 1988 - 1990

Teaching Assistant, Syracuse University, 1991 - 1997

Software Engineer, MNIS-Textwise Labs, Syracuse, NY, 1998 - 2002

Teaching Assistant, Syracuse University, 2003 - 2013 (intermittent)