University of California

Los Angeles

From Validation to Automated Repair & Beyond with Constraint Solving

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Hesam Samimi

2013

Abstract of the Dissertation

From Validation to Automated Repair & Beyond with Constraint Solving

by

Hesam Samimi

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2013

Professor Todd Millstein, Chair

Tremendous amounts of software engineering efforts go into the validation of software. Developers rely on many forms of software validation, from unit tests to assertions and formal specifications, dynamic contract checking to static formal verification, to ensure the reliability of software packages. Traditionally, however, the benefits seem to stop there, at checking whether there are problems. But once problems have been detected, those spent validation efforts play no role in the challenging task of debugging those problems, a task which requires manual, time-consuming, and error-prone developer efforts.

The key insight of this dissertation is that we can leverage the efforts that developers currently put into the validation of software, such as unit tests and formal verification, to get software engineering benefits that can go beyond validation, including automated software repair. Validation mechanisms can be elevated to this status using modern constraint solving, a technology that is already in use for the purpose of formal verification of software.

I present three novel and practical instances of this idea, that I was able to identify by focusing on particular domains and scenarios. The first work, used in development, builds on unit testing as the most common form of validation, and exploits a constraint solving method to automatically fix a certain class of bugs in the source code (*offline* repair). The second builds on dynamic, specification-based validation as in assertions and contracts used during development and testing, and applies it to deployed software to make it robust to unforeseen run-time failures by falling back to constraint solving (*online* repair). Finally, I use specifications and constraint solving to improve an existing validation methodology in test-driven development, used to enable testing when part of the depended upon software is unavailable or hard to set up.

The dissertation of Hesam Samimi is approved.

Rastislav Bodík

Tyson Condie

Jens Palsberg

Alan Kay

Todd Millstein, Committee Chair

University of California, Los Angeles

2013

to Minoo & Masoud

for bringing me here

Table of Contents

List of Figures

List of Tables

Acknowledgments

Thank you, Todd Millstein, for embodying the ideal role of an advisor. I came into your lab without much clue of what research, writing, peer interaction, and programming languages were all about. I part your lab looking and feeling like a much different researcher and person, and am forever indebted to you for holding my hand to come this far.

I wouldn't have been in Todd's lab in the first place, if it wasn't for the seminar course he and Alan Kay co-taught in the spring of 2008. Alan, you took a blind faith on me when I had no clue what was what from what. Eventually, your faith turned into a desire in me to become a researcher who can do what you want. I have come this far because of the will to get there and never been inspired more by an individual. Thank you.

I also thank my other past and present Ph.D. committee members, Jens Palsberg, Ras Bodík, Tyson Condie, and Rich Korf, for giving me valuable feedback during my qualifying exam on how to keep my research focused.

Thank you, folks at Viewpoints Research Institute: Kim Rose, Yoshiki Ohshima, and other colleagues for your continuous support. I sincerely thank Alex Warth for his guidance and helping me tremendously with putting together this dissertation.

My work titled Plan B, in Chapter 5, was done with Todd Millstein, Ei Darli Aung, and inspired by conversations I had with Ted Kaehler.

The Declarative Mocking work, described in Chapter 6, is joint work with Todd Millstein, Rebecca Hicks, and Ari Fogel. Thanks to Alan Borning who encouraged this project and is just an awesome guy to work with.

Thanks to all my collaborators in the Php Repair work of Chapter 3: Max Schäfer, Todd Millstein, Frank Tip, Shay Artzi, and Laurie Hendren.

Finally, I thank my parents for giving up so much to enable me in this journey. I love you very much.

I am thankful for the financial support I was given during my Ph.D. years, three Teaching Assistantships from UCLA, as well as multiple funding from Viewpoints Research Institute.

Vita

2004	Engineering Intern, Intel Corporation Hudson, MA.
2005	B.S. in Electrical Engineering and Computer Science, University of California, Berkeley Berkeley, CA.
2005—2007	Design Automation Engineer, Intel Corporation Santa Clara, CA.
2008	Teaching Assistant, CS35L, Software Construction Laboratory (Unix) Computer Science Department, University of California, Los Angeles
2008 & 2013	Teaching Assistant, CS131, Programming Languages Computer Science Department, University of California, Los Angeles
2009	M.S. in Computer Science, University of California, Los Angeles Los Angeles, CA.
2010	Research Intern Microsoft Research India Bangalore, India
2011	Doctoral Symposium, PC Chair, European Conference on Object-Oriented Programming Lancaster, UK
2008—present	Research Consultant, Viewpoints Research Institute Glendale, CA.

Publications

Alan Kay, Ian Piumarta, Kim Rose, Dan Ingalls, Daniel Amelang, Ted Kaehler, Yoshiki Ohshima, Hesam Samimi, Chuck Thacker, Scott Wallace, Alessandro Warth, and Takashi Yamamiya. STEPS toward the reinvention of programming. *VPRI NSF Technical Report, TR-2009-016, 2009.*

Hesam Samimi, Ei Darli Aung, and Todd Millstein. Falling back on executable specifications. In *Proceedings of the 24th European conference on Object-oriented programming, ECOOP 2010, Maribur, Slovenia.*

Hesam Samimi and Kaushik Rajan. Specification-based sketching with Sketch. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs, FTfJP 2011, Lancaster, UK.*

Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland.*

Alan Kay, Ian Piumarta, Kim Rose, Dan Ingalls, Daniel Amelang, Ted Kaehler, Yoshiki Ohshima, Hesam Samimi, Bert Freudenberg, Aran Lunzer, Alan Borning, Bret Victor, and Takashi Yamamiya. STEPS toward expressive programming systems, a science experiment. *VPRI NSF Technical Report, 2012.*

Hesam Samimi, Rebecca Hicks, Ari Fogel, and Todd Millstein. Declarative mocking. To appear in *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, Lugano, Switzerland.*

Chapter 1

1 Introduction

Tremendous amounts of software engineering efforts go into the validation of software. Developers rely on many forms of software validation, from unit tests to assertions and formal specifications, dynamic contract checking to static formal verification, to ensure the reliability of software packages. Traditionally, however, the benefits seem to stop there, at checking whether there are problems. But once problems have been detected, those spent validation efforts play no role in the challenging task of debugging those problems, a task which requires manual, time-consuming, and error-prone developer efforts.

The key insight of this dissertation is that we can leverage the efforts that developers currently put into the validation of software, such as unit tests and formal verification, to get software engineering benefits going beyond validation, including automated software repair. Validation mechanisms can be elevated to this status using modern constraint solving, a technology that is already in use for the purpose of formal verification of software. In my research I have strived to identify specific domains and scenarios in software engineering where combining existing validation mechanisms with modern constraint solving can lead to new practical software engineering benefits, both inside and outside of validation.

I present three novel instances of this idea in the areas of software development, debugging, and reliability. In each case I build on existing validation practices and deploy a practical constraint solving approach that results in additional software engineering benefits of reduced efforts in development and debugging, or increased reliability of deployed software. Additionally, I address the problem of making each approach practical by applying it in a specific scenario or application domain.

The first two works describe how to go from traditional validation practices to automated repair with the help of a constraint solver, while the third work demonstrates how this technology can improve upon an existing validation methodology itself.

Let me briefly overview these projects in some detail and then outline the organization of this dissertation.

1.1 PHP REPAIR: Unit Tests for Offline Repair

Unit testing remains as the prevailing form of software validation. More rigorous, formal verification approaches based on logical specifications may not scale well in practice in many scenarios, and average de-

```
1  <html>
2  <head>
3    <title>List capitals</title>
4    <style type="text/css">
5      .highlight { background-color: Aquamarine; }
6    </style>
7  </head>
8  <body>
9    <table border="2">
10   <tr><th><h3>Country</h3></th><th><h3>Capital</h3></tr>
11   <tr><td><div class='highlight'><b>Canada</b></div></tr></td>
12       <td>Ottawa</td></tr>
13   <tr><td><div class='highlight'><b>Netherlands</b></div></tr></td>
14       <td>Amsterdam</td></tr>
15   <tr><td><div class='highlight'><b>USA</b></div></tr></td>
16       <td>Washington</td></tr>
17   </table>
18 </body>
19 </html>
```

Figure 1: Fixing static HTML pages is (relatively) easy. Replaced or added tags are shown in red.

velopers lack the background and expertise to use them. Furthermore, in many domains unit tests more naturally express the desired behavior of software. The class of web applications used to generate HTML are an example of such a domain: while the inputs influence the output, the output is not really a direct function of the input, but rather determined by a combination of the page's design, wellformedness properties of HTML documents, information retrieved from a database, as well as the input parameters. In this context expressing the program's behavior in terms of simple input and output example pairs is much simpler than specifying logical formulas relating the output to all those that influence it.

Unit tests are used to reduce the burden on the developers to verify the code. But once bugs are found, can they also be used to help reduce debugging efforts? In the first part of my thesis I explore how existing unit tests, particularly in the domain of web programming, can be utilized as a basis for automated debugging during development.

We observe that a static malformed HTML page is relatively straightforward to fix (e.g. Fig. 1), but it is typically much more challenging to decide how to fix the generating PHP program, since a program's behavior is dependent upon dynamic information, control structure, and so on (e.g. shown in Fig. 2). I will later introduce the PHP REPAIR approach, which assigns the developer the less burdensome task of providing the corrected outputs for the failing tests, and in return automatically determines how to propagate the fixes applied to the outputs back to the generating PHP program.

```
<html>
<head><title>List capitals</title>...</head>
<body>
  <?php
    $highlight = isset($_GET["hl"]);
    $con = mysql_connect("localhost", "test", "test");
    mysql_select_db("countries", $con);
    $data = mysql_query("SELECT * FROM countries");
    if(!mysql_num_rows($data)) echo "<h1>No data!</h1>\n";
    else {
  ?>
  <table border="2">
  <tr><th><h3>Country</h3></th><th><h3>Capital</h3></tr>
  <?php
      while($row = mysql_fetch_array($data)) {
        echo "<tr><td>";
        if($highlight) echo "<div class='highlight'><b>";
        echo $row['country'] . "</b>";
        if($highlight) echo "</div></tr></td>"; else echo "</td>";
        echo "<td>" . $row['capital'] . "</td></tr>\n";
      }
      echo "</table>";
    }
  ?>
</body></html>
```

Figure 2: But fixing the generating PHP script is not as easy, due to the existence of program variables, control structures, and their dependence on dynamic information.

1.2 Plan B: Specifications for Online Repair

> "*Imagine a brain that consists of two parts, A and B. What he [Marvin Minsky] calls the 'A-brain' has inputs and outputs connected up to the real world. This can react to events in the real world; it 'thinks' in the real world. The 'B-brain', by contrast, has no direct connections to the real world; it only has connections into the A-brain. It 'thinks' in the domain of the A-brain—the A-brain is the B-brain's world! The B-brain's job is to 'correct' behavior in the A-brain, without any actual 'understanding' of the A-brain's thinking.*"
>
> —Headbirths blog [Blo], on Minsky's notion of B-brain in "Society of Mind [Min86]"

While significant amount of progress has been made in the area of formal software verification, we are nowhere near an era where shipped software packages are formally proven to work properly under all circumstances. As I mentioned earlier, most practices are limited to unit testing. In cases where formal methods are used, they are often cut down to checking only certain properties due to difficulties in scaling these techniques. As a result, software bugs and crashes continue to plague deployed software systems with costly ramifications.

In my work, called PLAN B, I implement Minsky's B-brain concept, as described above, as an automatic, run-time fault recovery system in Java programs. I demonstrate that if we take the implementation of a computer program as *Plan A*, it's full or partial formal specification, typically described in some form of logic, can play the role of *Plan B* at run time.

We can view the specifications as the "what" of the software, while the implementations (of an algorithm to achieve what is desired) are the "how." Traditionally specifications (the *whats*) have been employed towards the validation of the code (the *hows*), that is, simply as a way to *check* whether the code works properly. I show that by employing a constraint solver at run time, specifications can not just check, but also *correct* the behavior of buggy software, thus taking the role of B-brain as Marvin Minsky described above. Instead of halting the program on specification violations or run-time failures, we can fall back to constraint solving to automatically alter the run-time program state so that these specifications are satisfied, in order to allow the execution to safely continue.

Therefore, this work proposes to employ executable specifications for *online* repair, that is repairing the run-time state of a program rather than *offline* repair of the source code. While online repair introduces run-time overheads, it can be applied to deployed software, and is considerably more feasible because the constraints are solved for a particular execution of program and with known inputs, rather than for all possible executions.

I designed and implemented a concrete instantiation of the PLAN B approach as an extension to Java called PBNJ (PLAN B in Java). PBNJ augments Java with support for checking and automatically enforcing object invariants and method postconditions in a first-order relational logic similar to the Alloy [Jac02] modeling language.

As an example, Fig. 3 shows a square root function implementation on integers and an associated specification. The implementation in the example has an error as shown, causing an incorrect value to be returned. Let's assume the code gets shipped with this bug. As the method fails its dynamic postcondition check, the PBNJ runtime takes over and invokes the KODKOD constraint solver [Tor09] to find a model for the postcondition. The resulting model is translated into appropriate updates to program state variables (in this example the return value), and execution can safely continue.

The PLAN B system can be used in two ways. It can be deployed for *accidental fallback*, to automatically recover from unforeseen failures due to errors that escape the validation process and creep into the shipped software. The approach complements the static and dynamic verification techniques, and employs functional

```
int intSqrt(int x)
    ensures result >= 0 && result * result <= x
        && (result + 1) * (result + 1) > x  {
    int v = 1;
    int i = 1;
    while (v <= x) {
        v = v + 2 * i + 1;
        i = i + 1;
    }
    return i;      // <-- Bug: should return (i - 1)
}
```

Figure 3: A specification for the integer square root and its faulty implementation

specifications for recovering from software faults whenever they occur. It can be deployed in critical systems that are required to be robust in the face of occasional faults and crashes.

In the second mode, the developers may purposely omit parts of the code and use Plan B to execute specifications declaratively by default. I call this usage *intentional fallback*, which is particularly useful as an alternative to computationally expensive algorithms that are hard to code but simple to specify, such as search and constraint solving tasks. For example, a programmer could implement the common cases of an algorithm (e.g. normal inserts in a red-black tree) efficiently but explicitly defer to the specification to handle the algorithm's complex but rare corner cases (e.g. insertions in a red-black tree requiring rotations).

1.3 Declarative Mocking: Specifications as Mock Objects

I mentioned that Plan B can be used intentionally to perform declarative execution by default. However, in the majority of cases using constraint solving at run time is far too costly and inefficient as the main mode of operation. As stated, intentional fallback is only cost-effective if the task is computationally complex. In the last part of my research, I investigated whether or not declarative execution can provide realistic benefits in software engineering other than as a fallback mechanism described earlier to dynamically recover from code failures. I found an answer in the area of test-driven and agile development.

A common scenario during software development, especially during the early stages, is that the code under test depends on software components that are not available or difficult to set up. In the final part of this dissertation I argue that since logical specifications are directly executable in PBnJ, they can be used as *mocks*. That is, they can fill in the places of unavailable software components, thus enabling testing of code which depends on them.

```
List<Integer> currentProcessIds()
    ensures !(result == null || result.isEmpty());
```

Figure 4: Mocking a library method to produce a list of integers

The key idea is to leverage the fact that although the code under test needs to interact with pieces of software that are not there, typically the interface (API) and intended behavior of those are known. PBnJ specifications enable the developers to describe and obtain the intended behavior of components whose implementations are missing in a flexible and precise manner. The main advantage of the approach is the flexibility of logical specifications. For example, it is entirely up to the developer how partial or complete the specifications for the mocks are with respect to the real software components based on the particular needs in the individual tests.

As an example, assume we would like to test a GUI program that displays a table where the first column shows the ID of each currently running process on the system. The test depends on a system library method that is expected to produce an integer list representing those IDs. For the sake of this example let us say that this library is unavailable. How can we go about testing our GUI program nonetheless?

Note that for the purpose of this test, we do not particularly mind if the numbers in the table do in fact reflect the IDs of the real processes currently running. In Fig. 4 I show how a developer can use PBnJ specifications to enforce that a call to this method returns a representative non-empty list, which can be used in the test. I call the resulting approach *declarative mocking*.

Traditionally, software engineers have used *stubs* for the purpose of dealing with software components unavailable for testing. Stubs hard-code the outcome of an interaction with a mock. For instance, in the example above, we may stub the result of a call to `currentProcessIds` to return the list `[1272, 44425]`. I demonstrate that specifications and declarative execution can enhance the benefits of stubs. In particular, composition, underspecification, and nondeterminism, naturally expressible with logical specifications, can be exploited to have more flexible, dynamic, and reusable mocks, resulting in richer, higher coverage tests without requiring significant amounts of additional developer efforts. In addition to mocking functionality, this approach seamlessly allows data and other aspects of the environment to be easily mocked.

1.4 Thesis Statement and Organization

My thesis states the following.

We can combine development efforts that currently go into the validation of software, such as unit testing, executable specifications, and formal verification, with modern constraint solving technologies to obtain new practical software engineering benefits that go beyond validation. These advantages span several aspects of software engineering, from development and debugging to fault tolerance.

This dissertation demonstrates three different successful instances of the idea, strongly suggesting there are many other contexts and scenarios where declarative programming and constraint solving tools can be successfully deployed in software engineering practice.

The dissertation is organized into five main chapters.

- Chapter 2 provides the background for the dissertation. I discuss the traditional validation practices in software engineering which I build on in this thesis. Following that I overview the constraint solving technology that will be used to leverage those practices for the applications that are presented in the subsequent chapters.
- Chapter 3 focuses on automatic program repair during development based on unit testing, and in particular demonstrates a string constraint solving approach to automatically patch web applications that are used to generate HTML.
- Chapter 4 describes the PBnJ system, an extension of the Java programming language that I developed to support automatic, run-time checking and enforcing of executable specifications.
- Chapter 5 describes Plan B, the application of PBnJ to make deployed software robust to failures.
- Chapter 6 introduces declarative mocking, the application of PBnJ during development and testing as an enhancement to traditional mocks.

Chapter 2

2 Background

To better explain the works that will be presented, in this chapter I overview the current practices in software validation, as well as the state-of-art constraint solving tools that I use to build on those methodologies. I will describe the traditional approaches in validation only to demonstrate in subsequent chapters how they may be leveraged towards other goals related to software engineering spanning testing, debugging, and fault tolerance.

2.1 Current Software Validation Practices

Developers use a variety of approaches to check for the validity of software before shipping it. Some of these as in unit tests are quite cheap and light-weight. Less common approaches like formal methods are more costly but provide stronger guarantees about the correctness of the software packages.

2.1.1 Testing

Testing is the prevailing form of software validation due to its simplicity. Typically tests can be thought of input and output examples; given a specific input to a program, the code is invoked to dynamically check if the output agrees with the expected result. The obvious down-side of validation by testing is that nothing can be said about the correctness of the code under (often countless) other scenarios beyond what is covered in the test suite.

Chapter 3 will demonstrate a case where test suites can be leveraged not only to check for the existence of bugs in software, but also towards repairing them automatically.

2.1.2 Assertions, Executable Specifications, and Dynamic Contract Checking

Specifications have long been proposed as a means to express the intended semantics of a software component. The most common form of specifications are *assertions*—boolean expressions specified in the language of the code itself that state properties that must hold at that particular point in the program. As the program execution encounters an assertion, it dynamically evaluates it. Should result be *false*, a run-time assertion

exception is thrown, typically terminating the execution. In such a case, the run-time state of the program can be recorded and examined offline by the developer in order to debug the problem. Assertions may appear in unit tests as well as the programs themselves, intended to aid the developer in debugging the code.

Researchers have also realized that often the programming languages themselves are too low-level to compactly specify the behavior of programs written in them. Specification languages have thus been developed that allow specifying program semantics in terms of logic, typically in some form of first order logic over the state variables in a program. JML [LBR06] for Java and Spec# [BLS05b] for C# are two recent examples. The specifications, sometimes referred to as *contracts*, can express the same properties as in assertions, but also can appear as annotations in some context, e.g. method pre- and postconditions as well as object invariants that must hold after execution of every operation. In addition, they allow high-level, logical expressions such as quantifiers to enable more readable and manageable specifications. We saw an example of a postcondition specification in the PBnJ language for the `intSqrt` function presented in Chapter 1.

These assertions and contracts are often referred to as *executable specifications*, as these are expressions that can be evaluated dynamically by the language runtime into *true* or *false*. The specifications are typically used during development to dynamically (by contract checking [Mey97a] as described above) or statically (using program verification techniques [Lei07]) check that the implementation behaves as intended.

In Chapter 5 we will see how the Plan B system extends and applies this approach to deployed software. In a deployed system, halting the program on contract violations is no longer an option. The chapter proposes using a constraint solver at run time to alter the program state to meet the specification, allowing a failed execution to be rescued and safely continued.

2.1.3 Static Verification

Formal verification of software is the most desirable kind of validation, which involves statically examining the code and using logic to prove that a program meets its given specification under all possible executions. We mentioned previously that the practice is much less common. Not only it requires programmer expertise, its high computational cost may make it prohibitively inefficient to perform and scale up.

Static verification techniques employ constraint solvers as *theorem provers* to prove whether the code meets its specification. The approach goes as follows. Given the code of a program and its logical specification, we ask a constraint solver whether or not there exists some specific input on which the result of executing the code would violate the specification. In other words, we inquire whether the *negation* of the specification is

satisfiable. Unsatisfiability of such a case, if reported by the constraint solver, is essentially a proof of program correctness under all possible inputs. This kind of usage of constraint solving to check the correctness of software is sometimes referred to as *demonic* (i.e. asking "find me a bad case").

Chapter 4 will demonstrate how the same specification and constraint solving technology may be used at run time in deployed software in an *angelic* sense (i.e. asking "find me a good case"). Given a concrete run-time state and input to a program, we can enforce specifications by using a constraint solver to find a final state of the program in which the given specification is satisfied. Again, Chapter 5 employs this declarative execution system towards making deployed software robust to run-time failures.

2.1.4 Test-Driven Development, Stubs, and Mock Objects

A dilemma that developers frequently face during the early stages of development is that their code depends on other pieces of software that are not yet available. Software engineers have devised a technique, known as *mocking*, to deal with these situations.

There are mocking libraries (e.g. Mockrunner [AI], Mockito [Fab]) that allow the developers to automatically create *mock objects* [MFC01, FMPW04], which serve as dummy implementations of an API, enabling the programmers to write and test their code as if all the necessary dependencies are in place. By default invoking a method on a mock object has no effects, but typically a *stub* is set up in preparation for a given test to enforce a specific result from the invocation (recall the `currentProcessIds` example in Sec. 1.3 where a result `[1272, 44425]` was stubbed).

What makes mocking very attractive to software engineers is that it requires very little setup work, enabling them to easily run unit tests that otherwise would not be runnable without the presence of the dependencies. Moreover, since mocks mimic the real interfaces, once the missing parts become available virtually no code change is necessary to adapt to the real, instead of the mock, objects.

In Chapter 6 I show how executable specifications and constraint solvers can be leveraged on top of this test-driven methodology to produce mock objects that would lead to increased robustness, dynamism, and coverage of tests without requiring much additional developer efforts.

2.2 Modern Constraint Solving Technologies

There has always been a high interest in the application of constraint solving and logical decision procedures towards the analysis of software, as these enable the computer to use logic to automatically *think* and

reason about the behavior of a program. As I mentioned earlier, the problem is that these tools often are computationally very expensive (dealing with undecidable problems in general) and infeasible to apply in scales of realistic software systems.

During the past couple of decades the landscape somewhat changed with the emergence of highly efficient boolean satisfiability (SAT) solving algorithms (see e.g. [MMZ+01]).

2.2.1 SAT Solving

SAT is the logical problem of determining whether or not there exists an assignment of boolean variables appearing in a boolean propositional formula. SAT is the quintessential *NP-complete* problem, meaning in layman's terms computations that are very expensive to perform on a computer. One way to approach many other computationally hard problems, such as general constraint solving, is to show they can be reduced to pure SAT instances and then using a SAT solver to solve them.

In fact, today, many state-of-art constraint solvers are SAT-based. Modern SAT solvers, with the help of continued increase in micro-processor speeds (see Moore's Law), can now solve millions of boolean constraints in a matter of seconds. Consequently many modern constraint solvers simply encode the constraints in their domain into pure SAT to leverage the speed of the SAT solvers. This approach may reduce constraint solving times by many orders of magnitude, compared to when performing systematic search directly in the problem domain of constraints.

Next, I overview a SAT-based solving tool called KODKOD, which I employ in the works that will be presented in the future chapters.

2.2.2 Using KODKOD: an Off-the-Shelf SAT-Based Constraint Solver

This section overviews the main backbone for constraint solving used in all three works described in my thesis. KODKOD, developed by Torlak as her MIT PhD thesis [Tor09], is a SAT-based finite model finder for a first-order relational modeling language and algebra called Alloy [Jac02].

Alloy: Relational Modeling

Alloy is a modeling language (as in UML [RJB04]) that allows defining relations and expressing properties on them based on finite first order logic. Aside from standard logic constructs such as universal and existential quantifiers, relational operations like join and transitive closure are supported as well. This algebra has

been used to declaratively and compactly express models of hardware or software systems. The reason Alloy has been popular with software engineers and researchers has to do with its efficient automatic finite model finder—KODKOD—as its search backend. This backend can be used to formally and automatically verify or refute properties pertaining to a given model.

KODKOD: SAT-Based Solver for Alloy

KODKOD was developed as the search backend for the Alloy language, but it is now routinely used as a standalone constraint solving Java library. Given the description of a finite model universe and the relations involved, the user can specify properties about them in terms of first order relational logic of Alloy. KODKOD then encodes the constraints into a pure SAT problem and internally uses any off-the-shelf SAT-solver to efficiently find a concrete finite model in which the properties are satisfied, or else prove their unsatisfiability.

Controlling the Search Space: Lower and Upper Bounds

KODKOD supports the specification of *partial models*, a feature I heavily use in my thesis work. I briefly describe these here, but a complete explanation can be found in [Tor09].

Let's consider the finite universe `{a, b, c, d}` and a relation `r` over this universe. The tuples contained in each relation, drawn from the elements of the finite universe, may be explicitly specified. For example, we may specify `r(2): {⟨a, b⟩, ⟨b, b⟩, ⟨c, a⟩}`, meaning `r` is a binary relation containing exactly the three given tuples. In this case performing search does not involve exploring different possibilities for tuples within this particular relation, because they have been "fixed."

The tuples for a relation may also be specified as a search space, e.g. `r(2): [{}, {a, b} × {c, d}]`, meaning `r` is not forced to have any particular tuples (empty set on the left), but it may contain any tuples drawn for the pairs in the given cross product on the right. It is also possible to ask that a relation be a total function. E.g. `r(2): [{}, function({a, b} → {a, c, d})]` declares that `r` must contain exactly one pair where the first element is each of `{a, b}` and the second is drawn from the set `{a, c, d}`.

Specifying the tuples that a relation *must* contain is known as the *lower bounds*, and those that it *may* contain (select from) are the *upper bounds*. As we saw in examples above, we can define lower and upper bounds on a relation with syntax: `r(`*arity*`) = [`*lowerbound*`,` *upperbound*`]`

What distinguishes KODKOD from other constraint solvers such as SAT Modulo Theories (SMT) solvers (e.g. [DMB08]) is the ability to manually control the search space for the free variables in the constraints [1].

[1] In other tools this can be done only by specifying additional constraints. Yet this does not affect the "search space" for a variable and may not result in more efficient solving.

That is, both the lower and upper bounds may be specified. A set of lower bounds that partially specify the tuples that must be contained within their respective relations is known as a *partial model*. KODKOD can leverage partial models to reduce search space in the encoded SAT problem, significantly reducing solving times. This is one of the key features that I exploit to make constraint solving a practical option in the applications described in this dissertation. Let me demonstrate the usage of KODKOD through an example.

An Example: Sorting a List

Consider the following problem: given a linked list data structure `l1 = [0, -1, 2, 3, 2]`, find `l2` such that it is an increasing sort of `l1`. A finite universe relational model of this data structure is given in Fig. 5. Unary relation `Int` represents the set of numbers. Assuming 3-bit integers, these elements can be represented using 8 integers: -4, ..., 3. We will represent the `null` value as a singleton relation `Null`. We can use the next available integer—the number 4—to identify it. Each element in the linked list is represented a *node*, containing a *value* and *next*, a pointer to the linked node. We'll assume there are exactly 10 nodes in the universe (5 for each of `l1` and `l2`), so the unary relation `Node` can be represented using identifiers 5, ..., 14. Another unary relation `List` represents the two lists `l1` and `l2`, to which we assign identifiers 15 and 16. The entire universe is thus represented by integers -4, ..., 16 as shown in the figure.

Up until now all relations have been exactly specified. We will now move onto relations whose upper bound is larger than the lower bounds, i.e. there are more than one possibility for concretely choosing the tuples contained within those relations. A binary relation `Head` represents the head of each linked list. The head of list `l1` is known (say happens to be the node to which we assigned identifier 5), so the tuple $\langle$`15, 5`$\rangle$ appears in the lower bound. On the other hand `l2`'s head is unknown so it is not specified as a lower bound. The upper bound is specified as `function(List` $\rightarrow$ `(Node` $\cup$ `Null)`, since each list must have exactly one `Head` either set to `null` (empty list) or a node. The lower and upper bounds on binary relations `Next` and `Value` are set similarly, which represent the *next* pointer and *value* property for each node.

KODKOD can be invoked to obtain a model given these bounds. But we have not added any restrictions for list `l2`, so the model may choose any linked list value for it including empty list. Worst, it may choose an `l2` which is cyclic, i.e. there exist a cycle in the node links resulting in an infinite-size linked list. The next few lines add more restrictions on `l2`.

Line 73 uses universal quantifier, intersection, and non-reflexive transitive closure (`.^`) operations to specify the acyclic constraint: for every element `n` in unary relation `Node` the intersection of `n` and the transitive closure of `n` joined with `Next` relation must be empty.

```
58 // problem: l1 = [0,-1,2,3,2] and l2 = sort(l1). find l2.
59
60 // universe:
61 {-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
62
63 // relations:
64 Int(1): {⟨-4⟩, ⟨-3⟩, ⟨-2⟩, ⟨-1⟩, ⟨0⟩, ⟨1⟩, ⟨2⟩, ⟨3⟩}
65 Null(1): {⟨4⟩}
66 Node(1): {⟨5⟩, ⟨6⟩, ⟨7⟩, ⟨8⟩, ⟨9⟩, ⟨10⟩, ⟨11⟩, ⟨12⟩, ⟨13⟩, ⟨14⟩}
67 List(1): {⟨15⟩, ⟨16⟩}
68 Head(2): [{⟨15, 5⟩}, function(List → (Node ∪ Null)]
69 Next(2): [{⟨5, 6⟩, ⟨6, 7⟩, ⟨7, 8⟩, ⟨8, 9⟩, ⟨9, 4⟩}, function(Node → (Node ∪ Null))]
70 Value(2): [{⟨5, 0⟩, ⟨6, -1⟩, ⟨7, 2⟩, ⟨8, 3⟩, ⟨9, 2⟩}, function(Node → Int)]
71
72 // 1. lists are acyclic:
73 all n: Node | no (n ∩ n.^Next)
74 // 2. l2 is sorted:
75 all n: ⟨16⟩.Head.*Next | n.Next = Null or n.Value <= n.Next.Value
76 // 3. l2 is permutation of l1:
77 // 3a. l2 has the same number of elements as l1:
78 # (⟨16⟩.Head.*Next) = # (⟨15⟩.Head.*Next)
79 // 3b. the number of occurences for each element in l1 and l2 are the same:
80 all i: ⟨15⟩.Head.*Next.Value |
81     # { all n : ⟨16⟩.Head.*Next.Value | n.Value = i } =
82     # { all n : ⟨15⟩.Head.*Next.Value | n.Value = i }
```

Figure 5: Sorting a linked list `[0, -1, 2, 3, 2]` as specified for KODKOD

We also want `l2` to be sorted. This is specified in Line 75. We use join (`.`) and reflexive transitive closure (`.*`) operations to get the set of nodes in `l2`: `⟨16⟩.Head.*Next`. Any node `n` in this set must either have a *next* value set to `null` to have a *value* less or equal to that of the *next* node.

The specification of sort also requires `l2` to be a permutation of the list `l1`. We can state this property in two parts: Line 78 requires both lists to have the same number of elements (`#` is the cardinality operation). The second part establishes that for every value contained in list `l1`, the number of occurrences of that element is the same in both lists `l1` and `l2`. This is shown in Line 80, where the *set comprehension* syntax { `all` *var* `:` *set* `|` *condition*} is employed to get the set of occurrences of each element in both lists.

Given this set of constraints, a model can be obtained from KODKOD which this time properly represents `l2` as a sorted version of `l1`. One possible model is given in Fig. 6. Note that this model is not unique, since the solver is free to change either of the *next* or *value* properties for every node in order to sort the list.

```
Next:  {⟨5, 6⟩, ⟨6, 7⟩, ⟨7, 8⟩, ⟨8, 9⟩, ⟨9, 4⟩, ⟨10, 12⟩, ⟨11, 10⟩, ⟨12, 14⟩, ⟨13, 4⟩, ⟨14, 13⟩}
Head:  {⟨15, 5⟩, ⟨16, 11⟩}
Value:  {⟨5, 0⟩, ⟨6, -1⟩, ⟨7, 2⟩, ⟨8, 3⟩, ⟨9, 2⟩, ⟨10, 0⟩, ⟨11, -1⟩, ⟨12, 2⟩, ⟨13, 3⟩, ⟨14, 2⟩}
```

Figure 6: A model obtained by KODKOD for the linked list sort problem in Fig. 5

Chapter 3

3 Php Repair: Automated Repair of HTML Generation Errors in PHP Applications

Unit tests are ubiquitous in the software engineering practice, and so my initial direction is to explore how we can get more benefits from the vast efforts that programmers already put in producing these tests. Can the same unit tests that help finding bugs, also facilitate repairing them?

We know that synthesizing code is a very challenging task. Even today, programming is almost exclusively a human task. Our state-of-art tools can only generate very simple algorithms automatically. Automated repair of source code is equivalent to synthesizing part of a program, and so can be equally intractable. My research strategy is to identify and focus on specific domains and applications where automated repair of source code may be feasible.

This chapter presents a novel approach to automatically repair a certain class of bugs in the source code of programs whose main purpose is to generate text output. I focus on PHP applications, as the ever-present example within this domain, whose primary responsibility is to produce a text output on the server side representing an HTML page that would be sent to and viewed in the client's browser.

PHP web applications routinely generate invalid HTML. Modern browsers silently correct HTML errors, but sometimes malformed pages render inconsistently, cause browser crashes, or expose security vulnerabilities. Fixing errors in generated pages is usually straightforward, but repairing the generating PHP program can be much harder. I observe that malformed HTML is often produced by incorrect *constant prints*, i.e., statements that print string literals, and present a new tool for automatically repairing such HTML generation errors. Php Repair is a dynamic tool which, based on a test suite, encodes the property that all tests should produce their expected output as a string constraint over variables representing constant prints. Solving this constraint describes how constant prints must be modified to make all tests pass. This tool is implemented as an Eclipse plugin and evaluated on PHP programs containing hundreds of HTML generation errors, most of which the tools was able to repair automatically.

3.1 Introduction

PHP is the most widely used server-side programming language for implementing web applications, with a recent survey finding that it is employed by about 77% of all websites [W3T]. Typically, a PHP application generates HTML pages based on user input and information retrieved from a database. These HTML pages often contain JavaScript code to enable interactive usage and links or forms referring to additional PHP scripts to be executed.

One particularly common issue plaguing many PHP applications is generation of invalid HTML. Modern browsers are quite tolerant of HTML errors and employ heuristics to silently correct them, although pages may render more slowly because of these error-correcting heuristics [AKD+10]. In some cases, however, erroneous HTML will be displayed differently depending on the browser, so the pages generated by a PHP program may look fine to the developer, while they would be unacceptable to a user with a different browser. In extreme cases, invalid HTML may even cause browsers to become unresponsive or expose security vulnerabilities [AKD+10]. Finally, erroneous HTML can be an obstacle to screen readers and other assistive technology. The World Wide Web Consortium maintains a collection of anecdotes from Web professionals about problems with malformed HTML[2].

This chapter presents an approach to help programmers find and fix HTML generation errors in PHP programs. The approach is based on the observation that malformed HTML is most often generated due to errors in statements that print string literals. This is not surprising because such *constant prints* are the way in which a PHP program typically generates the tag structure of an HTML page. While, for example, the data in an HTML table might be generated via a dynamic database lookup in the PHP program, the table's HTML tags would be produced by printing the appropriate string literals (e.g., `"<tr>"` and `"<td>"`) in the right places.

An existing static tool called PHP QUICKFIX [SSA+12] can identify and fix shallow bugs in PHP programs, whereby a single constant print statement cannot possibly result in legal HTML. Examples include uses of HTML special characters such as `'&'` that should be escaped, or mismatched start and end tags as in `"<b><i>Yes!</b>"`. There is no need for any tests to identify these shallow bugs and PHP QUICKFIX statically examines the code to suggest quick-fixes to the user within a plugin for the Eclipse PHP Development Tools.[3]

I describe a new tool for fixing more general HTML generation errors which cannot be identified by examining a single constant print in isolation. PHP REPAIR, which I have implemented in the same Eclipse plugin,

[2] See http://www.w3.org/QA/2009/01/valid_sites_work_better.html.
[3] See http://www.eclipse.org/pdt/.

targets bugs caused by the interactions among multiple print statements and bugs that require adding, changing, or removing such statements. Given a test suite for a PHP program along with the expected HTML output for each test, we encode the condition that actual and expected output agree for each test case as a string constraint over variables corresponding to constant prints in the program. A string constraint solver automatically provides a solution to our constraint, which PHP REPAIR employs to modify the program appropriately. The result is a repaired program that passes all tests in the given suite.

PHP REPAIR only considers insertion, modification, and deletion of constant prints in a program. Despite the limited form of such repairs, they are still quite expressive, since the constant prints can be arbitrary; for example, a constant print may be inserted at any location in the program and there is no bound on the length of the string that it prints. Our focus on constant prints allows PHP REPAIR to perform an exhaustive search over the space of possible repairs, ensuring both *completeness* and *minimality*; prior work on automated program repair typically lacks these properties [WNGF09, NNNN11].

In principle, a purely static analysis could give stronger guarantees than PHP REPAIR's test-based approach. However, in practice the dynamic nature of PHP would make such an approach difficult to scale to real programs in a manner that only detects actual errors and fixes those errors without introducing new HTML generation bugs. In contrast, a testing approach is practical and effective for two main reasons. First, prior work has shown how to automatically generate high-coverage tests for PHP programs [AKD+10]. Second, while fixing a PHP program to correct HTML errors on all possible execution paths is quite challenging, fixing an individual broken HTML page is usually relatively straightforward, often requiring nothing more than, e.g., inserting a missing end tag. Indeed, such fixes are automatically suggested by tools such as HTML Tidy,[4] or they can be obtained by querying the DOM representation of the page inside a browser, which reflects the automatic corrections performed by the browser's HTML repair logic.

Note, however, that our intention is to avoid relying on heuristics used by these tools, and let the user have the final say over how the corrected outputs should look like. Therefore, while we produce a high-quality test suite using existing tools in a fully automatic manner, we only use the associated automatically generated test oracle required by this approach as a starting point. In my proposed usage model, we'll ask the user to inspect the automatically generated oracle as produced by HTML Tidy or extracted from the browser's logs, and make the necessary manual adjustments until her or his intentions are fully represented in the oracle.

I have evaluated the tool on several real-world PHP programs, showing that many HTML generation bugs can be fixed by this approach.

[4] See `http://tidy.sf.net`.

After using PHP QUICKFIX to statically identify and remove shallow HTML generation bugs, PHP REPAIR was able to fix on average 86% of the remaining bugs, which justifies my focus on constant print statements. A repair is found within seven seconds on average, so the tools are suitable for interactive use.

The remainder of the chapter is organized as follows. In Sec. 3.2 I provide some background and introduce the tool in the context of a motivating example. Sec. 3.3 precisely defines our notion of a repair and describes how the test-based tool PHP REPAIR finds repairs. Sec. 3.4 provides implementation details for the tool. Sec. 3.5 evaluates the tools on a set of PHP programs, Sec. 3.6 discusses related work and Sec. 3.7 concludes this chapter.

3.2 Background and Overview

3.2.1 An Example PHP Program

Fig. 7 shows a small PHP script designed to illustrate my approach. The program queries a database for a list of countries and their capitals and renders this data as an HTML table, optionally highlighting country names by printing them in bold face on a light blue background.

A peculiar feature of PHP is that programs can contain fragments of inline HTML code that are printed verbatim when the program is executed. In the program of Fig. 7, there are several such fragments; the first one (lines 84–91) prints the page header including a CSS stylesheet, while the last one (lines 116–117) prints the page footer.

Snippets of PHP code appear inside `<?php ... ?>` directives. The first snippet (lines 92–100) performs initialization and error checking: it uses the built-in function `isset` to determine whether the script was passed an HTTP GET parameter `hl`, setting flag `$highlight` accordingly; it then connects to a MySQL database containing the information to be displayed and sends a query to the database (lines 94–96).

If the query fails or returns no results, the body of the generated HTML page consists of the error message printed on line 98. Otherwise, another inline HTML fragment is used to emit the start tag of the table to be displayed (line 101) and its first row containing the column headers.[5]

To build the table, the script iterates over the results of the query using a `while` loop (lines 104–113). For every query result, it prints a new row of the table, with two `td` elements containing the name of the country

[5]Note that this HTML fragment is printed as part of the `else` branch of the `if` statement on line 97, which is only closed on line 114 in another PHP snippet: PHP code and HTML fragments can be freely mixed without regard to syntactic nesting, and this is frequently done in real-world programs.

```
<html>
<head>
  <title>List capitals</title>
  <style type="text/css">
    .highlight { background-color: Aquamarine; }
  </style>
</head>
<body>
  <?php
    $highlight = isset($_GET["hl"]);
    $con = mysql_connect("localhost", "test", "test");
    mysql_select_db("countries", $con);
    $data = mysql_query("SELECT * FROM countries");
    if(!mysql_num_rows($data))
      echo "<h1>No data!</h1>\n";
    else {
  ?>
  <table border="2">
  <tr><th><h3>Country</th><th><h3>Capital</h3></tr>
  <?php
      while($row = mysql_fetch_array($data)) {
        echo "<tr><td>";
        if($highlight)
          echo "<div class='highlight'><b>";
        echo $row['country'];
        if($highlight) echo "</div></tr>";
        else echo "</td>";
        echo "<td>" . $row['capital'] . "</td>";
        echo "</tr>\n";
      }
    }
  ?>
</body>
</html>
```

Figure 7: A simple PHP script

Table 1: Test cases for the script in Fig. 7

ID	database	parameters	output
t_1	$\emptyset$	$\emptyset$	see Fig. 8
t_2	countries = {(Canada, Ottawa), (Netherlands, Amsterdam), (USA, Washington)}	$\emptyset$	see Fig. 9
t_3	same as for t_2	{hl $\mapsto$ "1"}	see text

```
118 <html>
119 <head>
120   <title>List capitals</title>
121   <style type="text/css">
122     .highlight { background-color: Aquamarine; }
123   </style>
124 </head>
125 <body>
126 <h1>No data!</h1>
127 </body>
128 </html>
```

Figure 8: Valid HTML generated by the script in Fig. 7 on test case t_1

and its capital, respectively. If the script was passed the `hl` parameter, the country name is additionally wrapped in a `b` element to typeset it in bold font, and a `div` element with class `highlight`, which the CSS style sheet on line 88 styles using an aquamarine blue background.

3.2.2 HTML Generation Bugs

This example program contains several bugs similar to issues encountered in real-world PHP applications, which cause it to generate invalid HTML in certain situations. We will consider three test cases as described in Table 1: t_1 runs the script on an empty database without setting parameter `hl`; t_2 uses a non-empty database containing information about the capitals of Canada, the Netherlands and the USA, but again does not set any parameters; and t_3 runs it on the same database as t_2 with parameter `hl` set to `"1"`.

In test case t_1, the program produces the HTML page in Fig. 8, which is syntactically correct.

In test case t_2, it produces the page in Fig. 9, which is not valid HTML: the first `h3` element on line 138 is missing an end tag, as is the `table` element on line 137.[6]

These two problems are silently corrected by modern browsers, allowing the page to display as intended. For instance, inspection of the DOM produced when this page is displayed in Google Chrome 13.0 shows that it

[6]Perhaps surprisingly, the missing end tag of the second `th` element on line 138 is not a problem: `th` is self-closing, hence its end tag is optional.

```
129 <html>
130 <head>
131   <title>List capitals</title>
132   <style type="text/css">
133     .highlight { background-color: Aquamarine; }
134   </style>
135 </head>
136 <body>
137   <table border="2">
138   <tr><th><h3>Country</th><th><h3>Capital</h3></tr>
139   <tr><td>Canada</td><td>Ottawa</td></tr>
140   <tr><td>Netherlands</td><td>Amsterdam</td></tr>
141   <tr><td>USA</td><td>Washington</td></tr>
142 </body>
143 </html>
```

Figure 9: Invalid HTML generated by the script in Fig. 7 on test case t_2

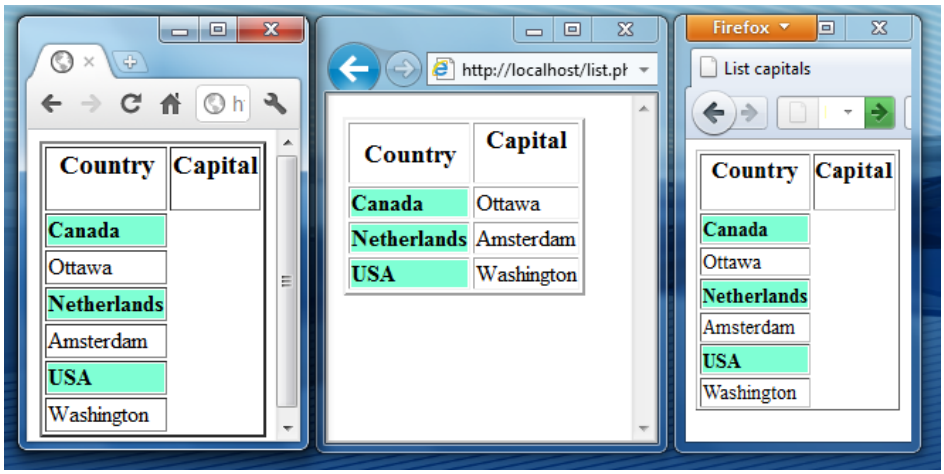


Figure 10: Different renderings of our invalid HTML page in Google Chrome 13.0 (left), Internet Explorer 9.0 (middle) and Firefox 6.0 (right)

inserts the missing end tags as expected: a `</h3>` tag before the `</th>` tag on line 138, and `</table>` before the `</body>` tag on line 142.

Translating these fixes for the generated HTML page into fixes for the generating PHP program can be much more difficult, however. It is clear that the first `h3` element in the inline HTML on line 102 must be closed by adding `</h3>` on that same line between `Country` and `</th>`. However, there are many possible options for inserting the missing statement `echo "</table>"` to close the start tag on line 101, and it is easy to do so improperly. If it is inserted as part of that same block of inline HTML, then the table will be closed before all of its rows have been output. If it is inserted inside the body of the while loop, the result on test t_2 will be to emit three `</table>` tags. Inserting it after line 114 would repair test t_2 but would break test t_1. Inserting it after line 113 leads to valid HTML being produced in both cases.

```
<html>
  <head>
    <style type="text/css">
      #test-div { margin:0 10px 10px; }
      #test-form { width:100%; }
    </style>
  </head>
  <body>
    <table>
      <tr><th>Test:</th></tr>
      <tr><td><div id="test-div">
            <form id="test-form" method="post">
              <input type="text" name="test"/>
          </div></td></tr>
    </table>
  </body>
</html>
```

Figure 11: An invalid HTML page that causes Internet Explorer to hang

Finally, consider test t_3, where the script is run on the same database as in t_2, but now with the parameter `hl` set to `"1"`. This produces essentially the same page as in Fig. 9, but the table rows are now of the following form:

```
<tr><td><div class='highlight'><b>Canada</div></tr>
    <td>Ottawa</td></tr>
```

This is not valid HTML: the `b` element is missing an end tag, and a `</tr>` tag occurs where a `</td>` tag is expected.

Different browsers display this invalid HTML page in different ways, as shown in Fig. 10. While they all insert the missing `</b>` tag before the `</div>` tag, the unexpected `</tr>` tag is not handled uniformly. Internet Explorer decides to treat it as a `</td>` tag, which is arguably the best fix from a user's perspective; Google Chrome and Firefox, on the other hand, silently insert a `</td>` tag before the `</tr>` and another `<tr>` tag after it, thus splitting one row into two and upsetting the table layout considerably. These kinds of inconsistencies, which can occur across different browsers as well as different versions of the same browser, are very easy for developers to miss during testing.

As before, propagating the desired HTML fixes (in this case, those performed by Internet Explorer) back to the generating PHP program is non-trivial. Of the various options for places to insert the missing `</b>` tag, the right place is on line 109 in the "then" branch of the `if` statement. Emitting it before the `if` statement would be acceptable for this test case but would break t_2. Propagating the other fix requires modifying the `</tr>` tag on the same line to `</td>`.

Problems such as incorrect or missing end tags may seem trivial, but they by no means always are: the HTML page in Fig. 11, adapted from `http://crashie8.com`, causes even very recent versions of Internet Explorer to hang while displaying without problems in other browsers. The problem here is a missing end tag for the `form` element starting on line 157 which, in a somewhat subtle combination with a table and a CSS stylesheet, triggers a bug in the browser's HTML repair logic. In some cases, invalid HTML can also impact browser performance, or lead to security vulnerabilities [AKD+10].

3.2.3 Automated PHP Program Repair

While repairing a PHP program can in principle require arbitrary modifications to its statements and structure, we observe that repairing HTML generation bugs often requires only additions, modifications, and removals of statements that print string literals. These include inline HTMLs or `echo` or `print` statements whose argument is a string literal, which I collectively dub *constant prints*. This is the case because, as illustrated in Fig. 7, constant prints are the mechanism by which the tag structure of an HTML page is generated. By focusing on this common class of repair actions, I have devised an approach to automatically repairing PHP programs that is simple yet effective, and have implemented the approach as a plugin for the Eclipse PHP Development Tools.

The fragile nature of PHP results in many shallow HTML generation bugs, whereby a single constant print is erroneous in the sense that it cannot possibly result in legal HTML, no matter what context it is executed under. An existing static checker tool PHP QUICKFIX [SSA+12] catches these kinds of local HTML generation bugs. Since this tool considers each constant print in isolation, it cannot detect or repair HTML errors involving multiple program points, such as the missing `</table>` tag in our running example. Generalizing PHP QUICKFIX to perform static analysis over an entire program would be quite difficult due to the many highly dynamic features of the language and the need to precisely model the effect of varying databases and parameter settings on control and data flow.

Instead, I propose a test-based approach to repairing complex HTML generation bugs. The approach assumes that a test suite for the program is available. Each test in this suite is described by: (i) the input data on which to run the program (such as HTTP parameters and databases), and (ii) the expected output the program is supposed to produce. Such a test suite can be produced without user interaction by employing a high-coverage test generation tool for PHP such as Apollo [AKD+10] and an HTML repair tool such as HTML Tidy, or it can be created manually. PHP REPAIR automatically adds, modifies, and removes constant prints in the given PHP program in order to produce a program that passes all tests in the given suite.

PHP REPAIR is based on the idea that we can characterize a given test's execution by the sequence of strings output by individual print statements that are executed in the program, say $s_1, \ldots, s_n$. If $s_1 \cdot \cdots \cdot s_n = e$, where "." represents string concatenation and "e" is the expected output, then the test case passes, otherwise it fails. Replacing each s_i that results from a constant print with a constraint variable v_i in the above equation encodes a string constraint whose solution tells us how to repair the program to satisfy the test case. A solution to all the constraints generated from a test suite leads to a repair that makes the whole suite pass.

Next I develop this basic idea in more detail on the basis of the examples we have seen in this section.

3.3 Input-Output Based Repair

3.3.1 Test Cases and Repairs

A program p is a collection of PHP scripts. A *test case* $t = (\rho, \sigma)$ consists of a configuration ρ to run the subject program under and an expected output σ. For the purposes of this discussion, the precise structure of ρ is irrelevant; it could, for instance, specify an initial database configuration, a sequence of scripts to execute, and the values of HTTP parameters to pass to the scripts. The *actual output* p produces on t is the HTML page generated by the last script invoked when running p under ρ.[7] The program is said to *pass* test case t if the actual string output of p on t equals the expected output σ.

For brevity let's refer to an inline HTML fragment or a `print` or `echo` statement as a *print,* and a print whose argument is a string literal a *constant print*, or *cprint* for short. Any other print is called a *variable print* or *vprint*.

Programs p and p' are called *repair convertible* if one can be obtained from the other by repeatedly performing any of the following repair actions: (i) adding a new *cprint*, (ii) removing a *cprint*, or (iii) modifying an existing *cprint* (by changing the string constant that it prints). A *repair problem* consists of a program p and a set T of tests. A solution of the repair problem is a program p' such that p and p' are repair convertible and p' passes all tests in T.

Note that we only consider repair actions involving *cprint*s. In particular, we do not consider adding, deleting or modifying *vprint*s, or changing the program's control structure. The evaluation in Sec. 3.5 suggests that most real-world HTML generation bugs can be repaired using only *cprint* repairs.

[7] While earlier scripts do not directly contribute to the actual output, they may alter the database or session state, and hence indirectly influence it.

```
<html>
<head>
  <title>List capitals</title>
  <style type="text/css">
    .highlight { background-color: Aquamarine; }
  </style>
</head>
<body>
  <table border="2">
<tr><th><h3>Country</h3></th><th><h3>Capital</h3></tr>
  <tr><td>Canada</td><td>Ottawa</td></tr>
  <tr><td>Netherlands</td><td>Amsterdam</td></tr>
  <tr><td>USA</td><td>Washington</td></tr>
  </table>
</body>
</html>
```

Figure 12: Expected output for test t_2 (non-empty database, no parameters)

As an example of a repair problem, consider the program of Fig. 7 and the test suite $T = \{t_1, t_2, t_3\}$ consisting of the test cases described in Table 1, which each only invoke a single script (the one shown in Fig. 7). The expected output for t_1 is the same as the actual output, shown in Fig. 8; the expected outputs for t_2 and t_3 are given in Fig. 12 and Fig. 13.

Fig. 14 shows the repairs to be performed to solve this repair problem, where changes are highlighted and unchanged portions of the program are omitted: two existing *cprint*s are modified, and one new *cprint* is added.

3.3.2 Properties

I have designed an approach that sets up a constraint system to capture the semantics of the repair problem as defined above, with solutions representing repairs. Before discussing it in detail, let us consider what properties we desire from such an approach.

1. *Soundness*: If the constraint system has a solution, it should represent a valid repair, i.e., the repaired program should pass every test in the suite.

2. *Completeness*: If a valid (*cprint*) repair exists, the constraint system should have a solution.

3. *Minimality*: For usability, we would like to find a repair that is minimal in the sense that it modifies the original program as little as possible.

```
<html>
<head>
  <title>List capitals</title>
  <style type="text/css">
    .highlight { background-color: Aquamarine; }
  </style>
</head>
<body>
  <table border="2">
<tr><th><h3>Country</h3></th><th><h3>Capital</h3></tr>
  <tr><td><div class='highlight'><b>Canada</b></div>
    </td><td>Ottawa</td></tr>
<tr><td><div class='highlight'><b>Netherlands</b></div>
    </td><td>Amsterdam</td></tr>
  <tr><td><div class='highlight'><b>USA</b></div>
    </td><td>Washington</td></tr>
  </table>
</body>
</html>
```

Figure 13: Expected output for test case t_3 (non-empty database, parameter `hl = "1"`)

```
...
102  <tr><th><h3>Country </h3> </th><th><h3>Capital</h3></tr>
...
109       echo " </b> </div></t d >";
...
114    echo "</table>";  }
...
```

Figure 14: Repair for the PHP script in Fig. 7

```
198 <html>
199 ...
200 <body>c1
201   <?php
202     ...
203     if(!mysql_num_rows($data))
204       echo "<h1>No data!</h1>\n"c2;
205     else {
206   ?>
207   <table border="2">
208   <tr><th><h3>Country</th><th><h3>Capital</h3></tr>c3
209   <?php
210       while($row = mysql_fetch_array($data)) {
211         echo "<tr><td>"c4;
212         if($highlight)
213           echo "<div class='highlight'><b>"c5;
214         echo ""c6;
215         echo $row['country']v1;
216         echo ""c7;
217         if($highlight) echo "</div></tr>"c8;
218         else echo "</td>"c9;
219         echo "<td>"c10;
220         echo $row['capital']v2;
221         echo "</td>"c11;
222         echo "</tr>\n"c12;
223       }
224       echo ""c13;
225     }
226   ?>
227 </body>
228 </html>c14
```

Figure 15: Labeled version of the script from Fig. 7

My approach makes two assumptions about the given program and its test suite. Firstly, the program may not inspect or modify its own source code; this is needed since we rely on source-level instrumentation to dynamically collect information about program executions. Secondly, all tests must be deterministic, i.e., the program must execute in the same way (and in particular produce the same output) every time it runs a given test. Since individual PHP scripts are not usually interactive this is not a severe restriction.

3.3.3 Finding a Sound Repair

Let a program p and a test suite T be given. If we assign a unique label to every print in p, we can characterize an execution of p on a test $t \in T$ by its *print trace*, which is the sequence of prints encountered during the execution together with the string values they printed.

Fig. 15 shows a possible labeling of the script from Fig. 7, where I have labeled the *cprint*s as c_1 to c_{14}, and the *vprint*s as v_1 and v_2. Multi-line HTML fragments are counted as a single *cprint* and only get a single

label. Additional empty *cprint*s have been inserted on lines 214, 216, and 224; these are necessary for the completeness of the approach and are explained in more detail below.

Using this labeling, the print trace of running the program on test t_1 is

$$[(c_1, \texttt{"<html>...")}, (c_2, \texttt{"<h1>...")}, (c_{14}, \texttt{"</body>...")}]$$

reflecting the fact that the program executed the *cprint*s on lines 198 - 200, 204, and 227 - 228 (in this order), but no *vprint*s.

Since a *cprint* will print the same string every time it is executed, we can abbreviate print traces by omitting the output of *cprint*s. Using this convention, the print trace of test t_2 is as follows (and is similar for t_3).

$$\begin{aligned}
&[c_1, c_3,\\
&c_4, c_6, (v_1, \texttt{"Canada"}), c_7, c_9, c_{10}, (v_2, \texttt{"Ottawa"}), c_{11}, c_{12},\\
&c_4, c_6, (v_1, \texttt{"Ne...")}, c_7, c_9, c_{10}, (v_2, \texttt{"Amsterdam"}), c_{11}, c_{12},\\
&c_4, c_6, (v_1, \texttt{"USA"}), c_7, c_9, c_{10}, (v_2, \texttt{"Washington"}), c_{11}, c_{12},\\
&c_{13}, c_{14}]
\end{aligned}$$

Clearly, the program passes a test case if the concatenation of all the output strings in the associated print trace equals the expected output. If we interpret the labels of *cprint*s as constraint variables, we can express the condition that actual output and expected output on a test case must agree as a string constraint: the left hand side of the constraint is the concatenation of all labels in the print trace, while the right hand side is simply the expected output. We will call this constraint the *repair constraint*.

The repair constraint for test case t_1, for instance, is

$$c_1 \,.\, c_2 \,.\, c_{14} = \sigma_1$$

where σ_1 is the HTML document of Fig. 8. One solution to the constraint has c_1, c_2, and c_{14} take on their original values, i.e., the string literals printed by the corresponding *cprint*s in the original program.

Since this approach to program repair only attempts to modify constant prints, we do not represent *vprint*s as constraint variables. Indeed, using a constraint variable for a *vprint* would in general lead to unsolvable constraints, since a single *vprint* may produce a different output each time it is executed (e.g., v_1 in Fig. 15).

Table 2: A solution for the repair constraints encoding t_1, t_2 and t_3

var	old value	repair value
c_3	`"...Country</th>..."`	`"...Country</h3></th>..."`
c_8	`"</div></tr>"`	`"</b></div></td>"`
c_{13}	`""`	`"</table>"`

Instead, we represent each occurrence of a *vprint* by the (constant) output it produced in the execution in question. For test case t_2, we then obtain the repair constraint

$$
\begin{aligned}
&c_1 \,.\, c_3 \,. \\
&c_4 \,.\, c_6 \,.\, \texttt{"Canada"} \,.\, c_7 \,.\, c_9 \,.\, c_{10} \,.\, \texttt{"Ottawa"} \,.\, c_{11} \,.\, c_{12} \,. \\
&c_4 \,.\, c_6 \,.\, \texttt{"Netherlands"} \,.\, c_7 \,.\, c_9 \,.\, c_{10} \,.\, \texttt{"Amsterdam"} \,.\, c_{11} \,.\, c_{12} \,. \\
&c_4 \,.\, c_6 \,.\, \texttt{"USA"} \,.\, c_7 \,.\, c_9 \,.\, c_{10} \,.\, \texttt{"Washington"} \,.\, c_{11} \,.\, c_{12} \,. \\
&c_{13} \,.\, c_{14} = \sigma_2
\end{aligned}
$$

where σ_2 is the HTML page in Fig. 12. This constraint, as well as the one above for test t_1, is solved by setting $c_3 :=$ `"...Country</h3></th>..."`, $c_{13} :=$ `"</table>"`, and all other variables to their original values.

A satisfying assignment for a set of repair constraints directly corresponds to a repair in which every *cprint* is modified to print the string assigned to its constraint variable. The repaired program will then by construction pass the test cases encoded by the constraints. Therefore, a solution to the repair constraints for a given test suite corresponds to a repair that makes the program pass every test in the suite. Table 2 shows a solution for the repair constraints encoding tests t_1, t_2 and t_3 (omitting unchanged variables), corresponding to the repair in Fig. 14.

3.3.4 Ensuring Completeness and Minimality

While the approach outlined so far is sound if so is the underlying constraint solver, it can only find repairs involving modifications of existing *cprint*s (including setting the string of a *cprint* to the empty string, which is tantamount to deletion). There is no support for adding new *cprint*s, hence the approach is not yet complete.

Note that it is never necessary to add a new *cprint* c' right before or after an existing *cprint* c: instead of adding c' we can just as well modify c. For the same reason, it is unnecessary to add c' if it is in the same block of straight-line code as c and there are no *vprint*s in between. Thus, the approach is complete if the

program to be repaired has a *cprint* at the beginning of every code block, after every *vprint*, and after every nested code block.

We can easily bring any program into this form by padding it with trivial *cprint*s of the form `echo ""`, and PHP REPAIR performs this simple modification. For instance, in the program of Fig. 15, *cprint*s are inserted on lines 214 and 224 after nested blocks, and on line 216 after a *vprint*. On the other hand, there is no need to insert a *cprint* after line 210 at the beginning of the loop body, as there already is a *cprint* on the next line.[8]

To achieve minimality, I use a cost metric to characterize the number of changes required by a repair. Let M be a solution for the set of repair constraints under consideration. Then we can define $cost(M)$ as the number of variables to which M assigns a different value than its original value, meaning that M is considered more expensive the more *cprint*s it modifies. In order to find a minimal repair for the program, we then simply look for a solution with the minimum cost. While this cost metric assigns the same cost to every modification, it is easy to substitute a different metric that, for instance, penalizes adding a *cprint* more heavily than modifying an existing one, or takes the amount of change to each print statement into consideration.

3.4 Implementation

In this section, I discuss implementation details of the PHP REPAIR tool, which I developed as an addition to a plugin featuring PHP QUICKFIX, built for the Eclipse PHP Development Tools (PDT).

Table 3: Diff regions for the example test suite

tests	actual output	expected output	diff variables
t_2, t_3	...Country </t...	...Country </h3> </t...	c_3
t_2, t_3	...</tr> </b...	...</tr> </table> </b...	c_{13}
t_3	...ada </tr> <td...	...ada </td> <td...	c_7, c_8
t_3	...nds </tr> <td...	...nds </td> <td...	c_7, c_8
t_3	...USA </tr> <td...	...USA </td> <td...	c_7, c_8

PHP REPAIR can be invoked via a menu item added to the PDT. The option requires the developer to specify an XML file which contains an encoding of the test suite.

PHP REPAIR first uses source-level instrumentation to generate the repair constraints. From the original program p it creates an instrumented program p_I that is identical to p except that trivial *cprint*s are inserted as described in Sec. 3.3.4 and all prints are replaced by calls to a logging function, which performs the normal

[8] An inserted *cprint* will not actually appear in the repaired program unless the solution to the repair constraints requires it.

print and logs both the label of the print and the output it produces. Running p_I on a test case produces a log containing the associated print trace, from which the repair constraint is constructed.

PHP REPAIR then solves the set of repair constraints by encoding them in the input language of KODKOD [Tor09], an efficient SAT-based constraint solver.

3.4.1 Why KODKOD?

In this formulation of repair we are dealing with string constraints, so I initially considered using an off-the-shelf string constraint solver such as Hampi [GKA+11] or Kaluza [SAH+10] instead. However, neither solver supports cost optimization, which we need in order to find a minimal repair. In contrast, KODKOD has an API for defining optimization problems, and can be easily used with any underlying SAT solver, including a cost-optimizing one.

Another advantage of KODKOD is that it provides a simple way to explicitly bound the allowed solutions to each variable, as we saw in the background Sec. 2.2.2. Such bounds are easy to obtain due to the simple form of our repair constraints, where the right-hand sides are constant. For example, the string value of a variable v appearing in a constraint C must be entirely composed of characters appearing on the right-hand side of C, and its maximum length is bounded by the length of the right-hand side.

I was able to use this feature of KODKOD to drastically reduce the search space, and consequently solve the constraints much more efficiently than when using the specialized string constraint solvers. This technique, along with an optimization that I will next describe, were key in making the approach scalable to real-world applications.

3.4.2 Other Optimizations

I further optimize the constraints passed to the solver by employing a simple *localization* heuristic, based on the observation that the differences between the actual and expected outputs for failing test cases are generally small. We first compute *diff regions* for each test case, i.e., substrings of the actual output that do not agree with the expected output. Using the logged print traces, we can then identify all *cprint*s that produce output in a diff region. I call the variables corresponding to these *cprint*s *diff variables*. In the example of Sec. 3.2, the actual and expected output on tests t_2 and t_3 yield five diff regions shown in Table 3. The last column lists the diff variables for every region; overall, the diff variables for this test suite are c_3, c_7, c_8, c_{13}.

Given this information, our heuristic forces all non-diff variables to retain their original values, since they do not contribute to any of the diff regions and hence likely already have their correct values. We do this by canceling out each non-diff variable from the left-hand side of each repair constraint, along with its corresponding expected output on the right-hand side (which by definition matches the variable's actual output). The result is a set of localized constraints in place of each original repair constraint. From the repair constraint for test t_2 we get three localized constraints

$$\begin{aligned} c_3 &= \texttt{"...Country</h3></th>..."} \\ c_7 &= \texttt{""} \\ c_{13} &= \texttt{"</table>"} \end{aligned}$$

whereas t_3 contributes only one new constraint:

$$c_7 \,.\, c_8 = \texttt{"</div></td>"}$$

These four constraints, with a drastically reduced search space than the original set of constraints, can be solved by KODKOD, leading to the solution shown in Table 2.

Localization is sound and critical in practice for reducing repair time, but it can sometimes lose solutions since it does not allow a *cprint* outside of a diff region to be modified. For example, consider a repair that requires hoisting a *cprint* from within an `if` block to occur just before the conditional. If the block is only executed on passing tests, our heuristic will not allow that *cprint* to be modified, causing the localized constraints to become unsatisfiable. We regain completeness (and thus the heuristic can be termed an optimization) through a simple back-off procedure: if the localized constraints are unsatisfiable, we expand each diff region by a fixed amount and try again. In the limit, each test output becomes a single diff region, causing the original repair constraints to be solved.

Finally, we observe that constraints that do not have any variables in common can be solved independently. We can hence improve constraint solving time by partitioning the repair constraints according to their variables and solving each partition separately. This optimization is particularly effective after localization, which tends to produce many constraints that each refer to a small number of variables.

Table 4: PHP REPAIR subject programs

program	version	# files	LOC	# tests	coverage
faqforge	1.3.2	19	734	536	89.2%
webchess	0.9.0	24	2,226	979	40.6%
schoolmate	1.5.4	63	4,263	676	65.5%
hgb	4.0	20	541	1359	97.2%
timeclock	1.0.3	62	13,879	958	26.8%
dnscript	N/A	60	1,156	1,167	75.9%

3.5 Evaluation

Let me now present an evaluation of our repair techniques on a set of PHP applications, focusing on the following evaluation criteria:

EC1 How successful is PHP REPAIR in repairing the HTML generation errors?

EC2 When PHP REPAIR fails, how often is this due to the restriction to modify only *cprint*s and how often due to inefficiencies in constraint solving?

3.5.1 Experimental Setup and Methodology

Table 4 describes the subject programs I used and their test suites. The LOC column lists the number of lines containing an executable PHP statement, and the last two columns give the size of the test suite and its line coverage. The tests were generated automatically using Apollo [AKD+10] on a time budget of 20 minutes. Coverage for *timeclock* is low since it makes heavy use of client-side JavaScript, which is not very well supported by Apollo. Note that each test typically triggers multiple HTML generation errors, and a single error may be triggered by multiple tests, so the number of failing tests tends to be correlated only loosely with the number of bugs.

PHP REPAIR additionally requires the expected output for each test. I used the W3C Markup Validation Service[9] to identify validity violations and HTML Tidy to automatically fix simple HTML errors. More complex errors that exceed the capabilities of HTML Tidy, as well as some of HTML Tidy suggestions I considered not matching the developer's intent, were fixed by hand. To address criterion EC2, we also manually constructed "golden" versions of the subject programs that produce the expected output on all tests. This required significant effort on the order of several days of work for the larger benchmarks.

On each benchmark, I first used PHP QUICKFIX to fix all the simple HTML generation errors and then applied PHP REPAIR to the modified program and its test suite to repair more complex errors. I believe this

[9] See http://validator.w3.org/.

approach reflects the way in which the Eclipse plugin featuring PHP QUICKFIX and my tool PHP REPAIR would be used by programmers.

To simulate a developer interacting with PHP REPAIR to repair all failing tests in a test suite $t_1, \ldots, t_n$, I used the following iterative process. Let t_f be the first failing test case. We first run PHP REPAIR on tests $t_1, \ldots, t_f$, with a timeout of three minutes for the solver. Recall from the end of Sec. 3.4 that we partition the repair constraints into independent sets. We automatically apply to t_f the repairs corresponding to each constraint set for which PHP REPAIR provides a solution. If all sets have solutions, then test t_f has been fully repaired so we move on to the next failing test case. Otherwise we manually apply as many fixes from the "golden" version as required to make t_f pass before moving on. We repeat these steps until all tests pass.

To measure the effectiveness of my approach we count the total number of *patches* (i.e., positions where a contiguous program fragment was inserted, modified, or removed) required to fully repair each program, and compute what percentage of patches were applied automatically by PHP REPAIR. This is a more objective metric than the number of fixed test cases, which depends heavily on the test suite. Using the number of validator error messages is also problematic, since a single error may lead to several messages.

3.5.2 Results

EC1: Table 5 reports on our evaluation of PHP REPAIR, listing for every benchmark the number of patches automatically applied by PHP REPAIR and the number of patches applied manually; the fourth column shows how many of the latter involved fixing statements other than *cprint*s and hence exceeded the capabilities of the tool. Across all benchmarks, PHP REPAIR on average performs 86% of all patches automatically. On these benchmarks, our iterative process went through a total of 125 iterations. On 42 iterations, PHP REPAIR timed out without finding a solution; the remaining iterations completed in an average of 7 seconds.[10]

In many cases for which PHP REPAIR timed out, the constraints were in fact unsatisfiable, implying the code required a patch outside the scope of the tool (e.g. a *vprint* modification or a change in the program's control structure). But, as is the case with most search-based constraint solvers, proving unsatisfiability with KODKOD is typically much more time-consuming than finding a model, when one exists.

The relatively low percentage of automated repairs on *hgb* is largely an artifact of our evaluation strategy: most manual patches could have been found automatically, but they occurred in the same test case (and constraint) as a more complicated (out-of-scope) repair and so had to be applied by hand.

[10] Measured on a 2.4GHz Core 2 Duo Macbook Pro with 2 GB of RAM.

Table 5: Number of errors found and repaired by PHP REPAIR

name	# tool patches	# all manual patches	# non-cprint manual patches	% tool patches
faqforge	33	3	3	92%
webchess	4	0	0	100%
schoolmate	11	0	0	100%
hgb	88	40	6	69%
timeclock	315	64	55	83%
dnscript	74	25	6	75%

EC2: The fourth column in Table 5 shows that on average only about 6% of the necessary repairs were out of scope for this approach. Examples of such repairs are missing `include` statements and faulty *vprint*s. The high number of non-*cprint* patches in *timeclock* is due to a single patch involving a *vprint* that is required in every script.

While most of the invalid HTML generated by our benchmarks would be silently corrected by a browser, I found three errors that resulted in visible layout problems, two of which were automatically fixed by PHP REPAIR.

3.5.3 Threats to Validity

The subject programs used in this evaluation may not be representative of other PHP programs. I did not specifically select the benchmarks to suit this approach; many of them have been used in prior test generation and fault localization research [AKD+10]. Some PHP programs (such as *phpBB2* [AKD+10]) use custom templating mechanisms to generate their output, whereby a template of the page to generate is read from a file and subjected to some string processing to generate the actual output page. My approach does not work well on such programs, which typically contain few cprints.

The bugs I detected and fixed may not be representative since the used test suites do not cover all of a program's behavior. However, the test suites achieve high coverage and were generated using algorithms that are completely unrelated to the repair techniques studied in this work.

Finally, there is often more than one way to fix a given HTML generation error, but in this evaluation I had to pick a single fix. When constructing the corrected HTML output and the golden versions of the subject programs, I have attempted to choose "sensible" repairs that disturb the original structure as little as possible.

3.6 Related Work

Static analysis of strings in web applications has been used to validate HTML output from web applications [Min05, MS11], to ensure that only XML documents meeting a given DTD are generated [MT06], and to detect security vulnerabilities [WGSD07, WS08, YAB11]. PHP REPAIR is only sound up to the given test suite. However, it can automatically repair HTML generation errors, rather than simply identifying them. Due to its dynamic approach, PHP REPAIR does not incur false positives as a static tool might.

Nguyen *et al.* [NNNN11] tackle the same problem of repairing HTML generation errors in PHP code, but in a very different way. They use a heuristic algorithm to map HTML output back to the program, while I use instrumentation to get a precise mapping. Like this work they focus on constant prints, but their heuristic repair algorithm does not appear to ensure soundness, completeness or minimality. Finally, their evaluation only considers fixes found by HTML Tidy; I also consider more complicated manual fixes, as I believe it is the developer, not the heuristic programmed in an automatic tool, that should define the right repair.

Weimer et al. [WNGF09] use genetic programming to repair C programs, whereby repairs are found by adapting statements from other locations in a program. Like my work, their approach requires a test suite, uses instrumentation to record execution paths, and guarantees correctness up to that suite.

My focus on constant prints allowed me to perform exhaustive search for repairs, ensuring both completeness and minimality. Genetic programming approaches support more complex repairs but rely on heuristics and hence lack these important properties.

Recent work by Meng *et al.* [MKM13] learns from sample repairs in the code to apply systematic edits to other locations in the code in a context/location-aware manner. Again their approach applies to a broader scope of repairs, but it requires sample repair to the code, whereas PHP REPAIR uses constraint solving to find repairs from corrected outputs rather than the code itself.

There has also been work on synthesizing programs that meet a given specification. Closest to this work are approaches that require the user to provide an initial program template with "holes" to be filled in [SLTB+06, SLJB08]. PHP REPAIR implicitly allows any *cprint* as a "hole" and uses tests to identify which ones to modify along with cost minimization to avoid unnecessary patches.

An example of such a work by Nguyen *et al.* [NQRC13] uses symbolic execution and program synthesis by constraint solving to automatically repair a certain class of expressions (arithmetic, array access, constants, etc.). This work is test-based as well and covers a broader scope for repair, but once again the down-side is

that there is no guarantee in finding such a repair, and it is unlikely that their tool can work on applications of the sizes comparable to those in benchmarks used in PHP REPAIR's evaluation.

Gulwani [Gul11] described a tool to synthesize Excel spreadsheet macros. Like PHP REPAIR, that approach is based on input-output examples and synthesizes a program that generates strings. However, programs are synthesized in a specialized domain-specific language, while my approach repairs arbitrary PHP programs.

Angelic debugging [CTBB11], like this approach, uses constraint solving over a test suite to identify erroneous expressions. While it can handle more general errors, angelic debugging is in general not able to suggest source-level repairs.

Several projects use constraint solving for automatic program transformations, often in the form of refactorings, as in type-related refactorings [TFK+11], refactoring for inferring generic types in Java [DKTE04], and refactorings that manipulate access modifiers [ST09].

3.7 Conclusions and Future Work

The chapter presented a novel approach to automatically repair HTML generation errors in PHP programs, targeting a common class of repairs based on adding, modifying, and removing statements that print string literals. I have developed test-based tool, PHP REPAIR, for repairing HTML generation errors by solving a system of string constraints. Our experiments show that these tools are able to efficiently repair most HTML generation bugs in a variety of open-source benchmark programs.

There are several avenues for further research. I would like to experiment with different cost metrics incorporating knowledge of the program's structure (e.g., to encourage solutions where all fixes are localized in the same script). To improve performance, we may be able to leverage the highly structured form of our constraints to aggressively optimize our SAT-based encoding, rather than relying on Kodkod's built-in encoding. I would like to generalize our approach to handle more complex repairs. And finally, another possible direction is to use similar constraint-based approaches to automate refactoring of PHP programs in an effort to reduce the accidental complexities of code due to developers' very often bad engineering habits.

Chapter 4

4 PBnJ: Declarative Execution in Java Using Kodkod

In Chapter 3 we worked with unit tests. In the rest of this dissertation, I turn to other forms of validation—assertions and formal specifications. Specifications are commonly deployed for the purpose of static formal verification, as well as dynamic contract checking. In static verification we often employ constraint solving to find bugs or prove the correctness of programs with respect to the provided specifications. Dynamic contract checking (e.g. assertions) is a complementary and more common validation practice, where more complex properties that are hard to statically verify can be checked at run time on a concrete execution of the program given a test input.

In Chapter 2 I proposed to use the same specifications and build on these validation mechanisms for purposes outside of validation. This chapter introduces PBnJ (Plan B in Java), an extension of Java programming language I developed that uses a constraint solver at run time to invoke specifications declaratively, that is, to non-deterministically alter the run-time state of an imperative program so that its given specification is satisfied. This method is sometimes referred to as *declarative execution*. PBnJ augments Java with support for checking and automatically enforcing object invariants and method postconditions in a first-order relational logic based on the Alloy [Jac02] modeling language.

Declarative execution in PBnJ is powered by Kodkod [Tor09], an efficient SAT-based relational constraint-solving tool, which can in turn invoke any standard boolean satisfiability solver to find models.

Later in Chapter 5 we will use PBnJ to enable online (run-time) repair when deployed programs crash or violate their specifications, and in Chapter 6 we use it along with executable specifications as a new form of mock objects.

4.1 An Overview of PBnJ

This section overviews PBnJ and its benefits through a motivating example. After illustrating PBnJ's specification language, I describe how these specifications are used for dynamic contract checking and declarative execution. Finally I discuss a few novel language mechanisms I have introduced to make declarative execution of specifications practical.

The PBnJ compiler is available at `http://www.hesam.us/planb` .

4.1.1 Specifications

Specification languages allow programs to be annotated with high-level specifications for the purpose of validation. For example, JML [LBR06] for Java and Spec# [BLS05b] for C# let developers annotate their programs, beyond the usual assertions, with object and loop invariants, pre- and postconditions, given in a variant of first-order logic.

PBnJ follows the same standard mechanisms for incorporating specifications into a Java program. Method postconditions are specified in an optional `ensures` clause on methods. Similarly, an optional `ensures` clause on a class declaration specifies any object invariants, which must hold at the end of the execution of each public method in the class. We have seen in Fig. 3 of Chapter 1 a small example, where an implementation of square root function on integers and its associated specification were given. As is common, the keyword `result` refers to the value returned by the method.

In addition to supporting side-effect-free primitive operations in Java, PBnJ's specification language includes a form of first-order relational logic based on Alloy [Jac02]. In this style, Java classes are modeled as unary relations (i.e., sets of objects), Java fields are modeled as binary relations between an object and its field value [11]. The syntax of `ensures` specification expressions is shown in Fig. 16 and includes forms of quantification as well as transitive closure on relations. I also provide procedural abstraction for specifications through a notion of *specification methods* (annotated in PBnJ as `spec`), which additionally support side-effect-free statement forms including assignment to local variables and if-then-else statements.

SpecExpr ::= *QuantifiedExpr* | *SetComprehension* | *SpecPrimary*
QuantifiedExpr ::= (`all` | `no` | `some` | `one` | `lone`) *QuantifiedPart*
SetComprehension ::= { (`all` | `some`) *QuantifiedPart* }
QuantifiedPart ::= *TypeIdentifier* [`:` *SpecPrimary*] | *SpecExpr*
SpecPrimary ::= *Lit* | *Primary* | *FieldClosure* | *IntegerInterval*
| *various Java primitive operations on integers and booleans*
Lit ::= `null` | `this` | `result` | *IntegerLiteral* | *BooleanLiteral*
FieldClosure ::= *Primary* `.(*` | `^` | `>)` *Identifier* (`+` *Identifier*)*
IntegerInterval ::= *Primary* `..` *Primary*

Figure 16: Specifications in PBnJ. The nonterminals ⟨Primary⟩, ⟨IntegeralLiteral⟩, and ⟨BooleanLiteral⟩ are defined as in the Java Language Specification [GJSB05]. See Alloy [Jac02] for semantics of quantifier types and relational operations.

Fig. 17 uses these features to provide the specification for a linked list implementation. The `List` class includes a `spec` method `nodes`, defined as the reflexive, transitive closure of the `next` relation starting from `this.head`. The `List` class uses this method to specify that the list must be acyclic (specification appearing at the class header) and to specify the postcondition for a sorting routine. Specification (`spec`) methods can

[11] One dimensional arrays are binary relations from index integers to values, and so on for higher dimensions.

```
class Node {
    int value;
    Node next;
}

class List ensures isAcyclic()  {
    Node head;

    spec PBJSet<Node> nodes() { return head.*next; }

    spec PBJSet<Integer> values() { return nodes().>value; }

    spec boolean isAcyclic() {
        return head == null || some Node n : nodes() | n.next == null;
    }

    spec boolean isSorted() {
        return all Node n : nodes() |
            (n.next == null || n.value <= n.next.value);
    }

    spec PBJSet<Node> nodesOfValue(int i) {
        return { all Node n : nodes() | n.value == i };
    }

    spec int occurrencesOf(int i) { return nodesOfValue(i).size(); }

    spec boolean isPermutedSublistOf(List l) {
        return all int i : values() | occurrencesOf(i) <= l.occurrencesOf(i);
    }

    spec boolean isPermutationOf(List l) {
        return this.isPermutedSublistOf(l) && l.isPermutedSublistOf(this);
    }

    void sort()
        ensures this.isPermutationOf(this.old) && this.isSorted();
}
```

Figure 17: A linked list of integers in PBnJ. The `sort` method is declarative, as it contains no code.

invoke other specification methods, but not ordinary Java methods, and only specification methods can be invoked from an `ensures` clause. The `nodesOfValue` method uses PBnJ's facility for set comprehension, and the `values` method uses the `.>` operator to map the `value` relation on each node in `nodes()`.

Each object in PBnJ has an implicit field named `old` that can be used in method specifications to refer to the state of that object on entry to the method. This simple mechanism is very powerful because `old` has a "deep copy" semantics. For example, the specification of `sort` uses the `old` field of `this` to ensure that the implementation of the method does not add or remove any nodes from the list. Because of the declared object invariants, the resulting list is also required to be acyclic.

A secondary benefit of specification methods in PBnJ is that they are directly executable. Specification methods can be invoked by ordinary Java methods and thereby used as part of the implementation of a class. For example, clients of our list can invoke the `nodes` method to get a set of all nodes which can then be manipulated as usual in Java code. In this way, specifications are useful not only as a declarative execution mechanism, but also to make implementations more declarative and hopefully more reliable by construction. The PBnJ compiler automatically translates specification methods into ordinary Java methods (see Sec. 4.2). I represent sets using a `PBJSet` class which implements the `Set` interface in the Java standard library but also provides functional versions of several operations (e.g., adding an element, set union), since specifications need to be free of side effects.

Note that the `sort` routine has no implementations, but only a postcondition specification. In PBnJ the routine is still runnable, because specifications are declaratively executable by invoking the constraint solver that is integrated in the language. In the next section I will describe this process.

4.1.2 Declarative Execution

Fig. 18 illustrates the execution of a declarative method, i.e. one that has specifications but no implementation, in PBnJ. The process starts with translating the method postcondition and any object invariants, as well as the state of program on entry to the method, into the logic of the Kodkod relational constraint solver [Tor09]. Details on this translation are provided in the next section. We then invoke the solver to search for a *model* satisfying the specification. If a model is found we use the model to transform the current program state and continue execution. If Kodkod reports unsatisfiability, then the specifications have no solution within the search bounds provided by the programmer (see Sec. 4.2.4). In such a case a `ContractViolationException` is thrown, similar to what would happen with traditional contract checking.

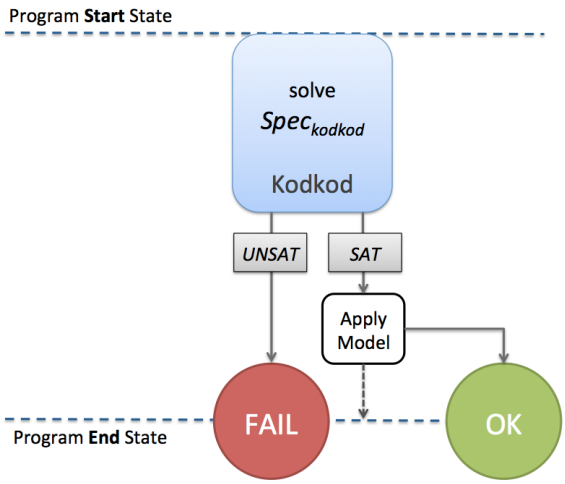


Figure 18: Declarative execution of specifications in PBnJ

I have implemented a compiler for PBnJ using the Polyglot extensible compiler framework [NCM03]. The next section details the compilation scheme.

4.2 Implementation

Specifications in PBnJ can be used in two ways. The user may just invoke a `spec` method, as an ordinary Java function with a boolean return value, to check whether a property is satisfied or not. As we saw before, in case the specifications are part of the postcondition of a method which has no code, they are declaratively executed by their translation into the underlying constraint solver Kodkod. In this section I explain how PBnJ specifications can be translated to both Java and Kodkod.

4.2.1 Translating Specifications to Java

The PBnJ compiler translates each specification method to regular Java code and creates a regular Java method for each declared method postcondition and object invariant. The translation from our specification language to Java is straightforward. For example, the transitive closure operation on a field f is implemented by a simple worklist algorithm that traverses f fields from the specified root object and adds each encountered object into the result set until reaching either the `null` value or an object that has already been encountered. The compiler then instruments the body of each declarative method in a class to invoke the the Kodkod solver on the specified postcondition and any related object invariants (described below).

```
Relation nodes_kk() {
    return
        This.join(List_head).join(Node_next.reflexiveClosure()).difference(Null);
}
```

Figure 19: KODKOD translation of the `nodes` specification method in Fig. 17

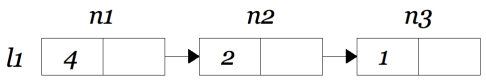


Figure 20: An example of the reachable state from the receiver object on entry to some invocation of `sort` from Fig. 17

4.2.2 Translating Specifications to KODKOD

The compiler also creates versions of each specification method, postcondition, and object invariant for input to KODKOD. Each of these translations is placed in a new method inside the enclosing class. KODKOD is implemented as a Java library [Tor09], so each method contains regular Java code that constructs a KODKOD-specific data structure representing the original specification formula. Dynamically these methods are invoked to build a data structure representing a KODKOD formula, which is passed to the solver for model finding.

Since our specification language is based on KODKOD's relational logic, these methods are quite simple. For example, Fig. 19 shows the translation of the `nodes` specification method from Fig. 17 to a method that constructs a corresponding KODKOD `Relation`. The compiler declares a unary relation corresponding to each class and a binary relation corresponding to each field. For example, Fig. 19 refers to `List_head`, which is a binary relation between lists and nodes. Accessing the field value of a particular object corresponds to a join of the (singleton) relation denoting that object with the field's relation. As the example shows, we also declare singleton relations to represent `this`, `null`, and other values in scope. The `null` value is removed from the result set to properly account for the semantics of transitive closure in PBNJ. Finally, to handle specifications that refer to `old`, the compiler creates a second binary relation for each field to represent that relation's value on entry to a method for which we perform declarative execution.

4.2.3 Model Finding with KODKOD

Consider the `sort` method in Fig. 17. Dynamically on entry to the method, we invoke the method `sort_spec_kk`, which produces a KODKOD `Formula` representing the postcondition. We do the same for the declared object invariant and conjoin these formulas. In order to find a model for the resulting formula, we

Table 6: A relationalized version of the program state in Fig. 20

relation	bound
List	{⟨L1⟩}
Node	{⟨N1⟩, ⟨N2⟩, ⟨N3⟩}
List_head_old	{⟨L1, N1⟩}
Node_value_old	{⟨N1, 4⟩, ⟨N2, 2⟩, ⟨N3, 1⟩}
Node_next_old	{⟨N1, N2⟩, ⟨N2, N3⟩, ⟨N3, Null⟩}

Table 7: Default bounds for the relations to be solved for in a declarative execution for `sort` from Fig. 17

relation	lower bound	upper bound
List_head	{}	function(List→(Node+Null))
Node_value	{}	function(Node→Integer)
Node_next	{}	function(Node→(Node+Null))

must provide KODKOD with *lower* and *upper* bounds for each relation. Recall that upper bounds specify which values *may* appear in a relation, while lower bounds state which ones *must* appear. The bounds for the unary relations representing classes as well as the binary relations representing *old* field values are created by traversing the reachable state starting from the receiver and formal parameters. For example, suppose the receiver object on entry to an invocation of `sort` looks as in Fig. 20. In that case, we will set both the lower and upper bounds for various relations as shown in Table 6, ensuring that KODKOD cannot change the values of these relations.

By default, the bounds on all other relations are trivial, as shown in Table 7. Each relation has an empty lower bound and an upper bound that simply indicates the type of the relation. The `function` keyword tells KODKOD that these binary relations are in fact functions from the first component to the second, which ensures that all solutions will be valid. For example, the upper bound for `Node_next` relation is requiring either a node or null value for each of the node objects shown in Table 6.

Once the bounds are calculated, we invoke the KODKOD solver to find a model satisfying the specification formula. If a model is found, we iterate through the relations of the resulting model and use reflection to update the corresponding objects' fields with the specified values. Program execution then continues.

4.2.4 Making Constraint Solving Practical

I have developed two main techniques to allow PBNJ programmers to declaratively bound the search space for declarative execution in an application-specific manner. Both these use KODKOD's feature to explicitly define lower and upper bounds on search spaces for the variables inside constraints (reviewed in Sec. 2.2.2).

Frame Conditions: By default, the constraint solver can modify the values of any fields (instance variables) mentioned in a postcondition or object invariants in order to perform declarative execution. However, it is useful to allow programmers to override this default by providing an explicit frame condition as a subset of fields that are intended to be modifiable. This is done with an optional `modifies fields` clause on a method. Traditionally such *frame conditions* have been used to ensure the absence of any disallowed updates to the program variables during the static verification of a method (e.g., [FLL+02]). Beyond this purpose, I use these annotations to improve the performance of constraint solving by limiting the search space. For example, annotating the `sort` method in Fig. 17 with the following clause prevents the solver from attempting to change the integer values stored in each list node, only permitting updates to their `next` pointers:

```
modifies fields List:head, Node:next
```

Of course, frame conditions are also used for the traditional purpose of simplifying a specification by ensuring that certain nonsensical solutions and disallowed updates are ruled out.

By default any object reachable from `this` and formal parameters on entry to the declarative method may be modified. I allow programmers to override this default through a novel `modifies objects` clause, which specifies a Java expression that evaluates to a collection of objects; PBnJ's declarative execution mechanism considers all other objects to be immutable for purposes of declarative execution. Consider again the `sort` method, and assume the enclosing `List` class also happens to have a reference to another `List` object, for an unspecified reason. But based on the `modifies fields` annotation above, this procedure would be allowed to modify the `head` field for *any* `List` object, and `next` fields for *any* `Node` object that it may have a reference to. We need to further restrict the search space for the declarative execution of `sort` to only allow modification of fields of `List` object `this`, as well as the set of `Node` objects contained in it.

Rather than building this constraint into the postcondition, which would be tedious and complex, the programmer can provide the following clause, where `nodes()` returns the set of contained nodes in our list and `.plus(this)` returns a new set that adds the list itself to this set.

```
modifies objects nodes().plus(this)
```

PBnJ invokes the `modifies objects` expression dynamically when a declarative execution event is triggered, and the resulting set of objects is communicated to the solver as being modifiable; all other reachable objects are treated as immutable. This approach allows for significant flexibility, as the frame condition can be an arbitrary expression evaluated only when declarative execution is about to occur.

Applying frame conditions like above results in a reduction in search bounds specified for the constraint solver Kodkod. For example, suppose the `modifies fields` declaration for `sort` in Sec. 4.2.4 is provided by the programmer. In that case we know that the `Node_value` relation should be unchanged in the solution. Therefore we use the value of the `Node_value_old` relation from Table 6 as both the lower and upper bound for the `Node_value` relation. Further suppose a `modifies objects` clause is provided by the programmer. In that case, we execute the associated expression on the receiver to obtain the set of modifiable objects. For any object o not in the result set, with relational counterpart atom O, for any pair (O, O') in some relation of the form `C_f_old`, we also place the pair (O, O') in the lower bound for the relation `C_f`. This ensures that while the relation `C_f` can be modified by KODKOD, the value of o's field cannot change in the solution.

Bounding the Universe: KODKOD is a SAT-based reasoning tool, and it therefore expects a finite bound for the search space for each of the types, including primitives. This implies that if the tool does not find a model, we can only assume the problem is unsatisfiable within the given bounds. This incompleteness can cause PBNJ to signal a contract violation exception when a satisfying program state might have been possible.

When executing specifications involving search for integer values, the usual 32-bit integer range is often not a tractable space for SAT-based solvers. By default PBNJ assumes 8-bit integers when executing specifications. However, we allow the user to explicitly set bounds for integers. The number of objects of each class must also be bounded. By default we bound each class by the number of instances of that class that are reachable from the receiver and formal parameters at the point of declarative execution. However, this bound is insufficient if the declarative method may need to instantiate new objects. To handle this situation, we allow each method to include an optional annotation specifying an upper bound on the number of new instances of each class that KODKOD may create in order to satisfy the method's specification. For instance, the specification for an `add` method in our `List` class, which adds one element to the list, states that one new `Node` object should be allowed:

```
adds 1 Node
```

Once again the number may be an arbitrary expression only determined at run time. If the method contains an `adds` annotation indicating the number of new objects of each class that may be created, for translation to Kodkod we update the associated unary relations with a corresponding number of fresh atoms.

4.3 Related Work

My tool builds upon several lines of research on executable specifications and software reliability.

4.3.1 Executing Specifications via Constraint Solving

PBnJ uses Kodkod [Tor09] to enable declarative execution. Couple of more recent declarative execution systems that may have been influenced by my tool include Squander [MRYJ11] for Java and Kaplan [KKS12] for Scala. Squander also uses Kodkod for finding models, but fully incorporates the Alloy relational modeling language [Jac02] and employs a clever Kodkod encoding scheme that can reduce constraint solving times. In PBnJ the specifications are expressed over concrete Java variables in the program. Squander, on the other hand supports abstract, logical variables that are used for specification purposes only. Abstraction and concretization functions can be then provided to relate the concrete and logical states of a given program. Kaplan utilizes the state-of-art SMT solver Z3 [DMB08] for constraint solving. Kodkod has a clear efficiency disadvantage compared to SMT solvers for problems involving primitive values such as integers, because it is based on direct translation to SAT, but SMT solvers utilize theory-specific decision procedures to avoid naive search. On the other hand, Kodkod allows relational operations such as closure and comprehension, which I enabled in this language, but Z3 does not. More importantly, Kodkod allows explicit setting of search space for variables, which I utilize for the `modifies` frame specifications. In most constraint solving tools, including Z3, exact bounds can only be set via additional constraints, which may not necessarily result in a smaller search space.

An advantage of PBnJ compared to most other recent mixed declarative-imperative languages like Kaplan, as well as library-based constraint solving tools like Gecode [STL], is that existing programs can use the constraints by simply being compiled by the PBnJ compiler. There are no additional steps for the user, beyond writing the constraints, to use the system. Based on the class declarations, the compiler automatically instruments the binary to make the necessary translations from the run-time program state into logic and invoke the constraint solving backend when needed. PBnJ has undergone many updates and upgrades mainly because, for the two of its applications that will be presented in Chapters 5 and 6, we had to perform a comprehensive evaluation of its usage on existing open-source applications. The compiler supports constraint solving for Java programs making use of complex but common features like generic types, nested classes, and others.

The idea to execute specifications has been explored in a variety of contexts. Two recent examples include work on executable specifications for C++ [WLB00] as well as for the JML modeling language for Java [KW06]. These works allow specifications to be executed on their own, for the purpose of gaining confidence in their correctness. My work also executes specifications but in a manner that is tightly integrated with the host programming language. Specifications in PBnJ are directly executable as part of a Java program's execution, and my notion of declarative execution requires constraint solving to happen online and in collaboration with ordinary program execution.

The idea of a *mixed interpreter*, which can execute programs that consist of both specifications and implementations, is more closely related to this work. Morgan laid the formal foundations for this approach with his notion of a *specification statement* [Mor88]. However, his goal was not to automate the execution of specifications but rather to support program reasoning uniformly during the process of manually *refining* specifications to implementations. Freeman-Benson and Borning introduced the notion of *constraint imperative* programming as embodied in their Kaleidoscope language [FBB92], which can be viewed as an instantiation of the idea of a mixed interpreter. This language allows a class to declare constraints that are automatically enforced on instances of the class using a constraint solver that integrates decision procedures for several domains. Rayside *et al.* have recently described a vision of "agile specifications," which employs the notion of a mixed interpreter to unify the benefits of formal methods with those of the agile software development methodology [RMY+09a]. In Chapter 6 I concretely develop some of the ideas they laid out in the paper.

4.3.2 Alloy

My use of a specification language adapted from Alloy [Jac02] and of Alloy's underlying solver Kodkod [Tor09] are no accident. Alloy's relational style of modeling programs has proven to be quite natural and powerful, and Alloy and Kodkod have been used in a variety of ways to gain confidence in the correctness of imperative programs. One line of work employs these tools for *bounded verification* of implementations [JV00, KMJ02, VJ03, DCJ06, DYJ08]. In this approach, both the body of a Java method as well as its specification are translated to Alloy/Kodkod, and a constraint solver searches for executions (up to some bound on the length of an execution trace and the size of the heap) of the method that violate its specification. The Analyzable Annotation Language (AAL) [KMJ02] additionally employs Alloy to reason about the specifications themselves, for example to ensure that the specification of an `equals` method in Java is in fact an equivalence relation. Finally, TestEra [KM04] uses Alloy as a test generation tool for Java

programs. Alloy generates non-isomorphic inputs up to some bound that satisfy a method's precondition, and Alloy is also used as the test oracle to determine whether the result of executing the test satisfies the method's postcondition.

My work borrows the basic relational approach to modeling objects from these prior works, along with the approach to translating a program state into relations for input to KODKOD. However, rather than using this technology to gain confidence in an implementation, this approach ignores the implementation and instead employs KODKOD to "execute" Alloy-style specifications. The approach avoids some complications of verification, for example the need to translate arbitrary Java code to relational logic. On the other hand, our use of online constraint solving poses a performance challenge, which I have addressed in part through novel program annotations, e.g. the `modifies` clause.

4.4 Discussion and Future Work

More needs to be done to make PBNJ industry-ready. First, the PBNJ language can be improved in a few important ways. The specification language lacks support for floats and strings and associated operations. This is due to our decision to use Kodkod, a bounded solver that directly encodes constraints into SAT, and which only supports integer variables among primitive types. Constraints involving other primitives such as reals, floats, and strings are not directly supported by Kodkod. In the current implementation of PBNJ we only support equality constraints on such values (via boxing).

There is no support yet for constraints over nested generic types (e.g. `List<List<Integer>>`), nor is there any special support for class inheritance currently. For example, the specification of a method is not automatically inherited by an overriding method and need not have any relationship to the specification of that method. Finally, specification methods are currently forbidden from being recursive, which limits their expressiveness. Others have shown how to translate recursive definitions into Alloy [KMJ02], so it would be natural for us to adopt their approach.

Additionally, the PBNJ implementation can be optimized to reduce the overheads of declarative execution. SAT-based constraint solving can become prohibitively slow when too many integer variables are involved due to the large search space. For this reason, I typically have run the constraints assuming only 8-bit integers.

Because of the above the most relevant update I would like to add to the tool is the ability to invoke solvers other than KODKOD. Most notably, utilizing an SMT solver like Z3 [DMB08] in the backend would enable

support for float and string primitive values inside specifications and greatly reduce constraint solving times on problems involving numbers or other primitives. This is because SMT solvers avoid naive search where theory-specialized solvers such as Simplex for linear arithmetic constraints or bit-vector solvers for string constraints are available.

Lastly, I am interested in enabling support for other modes of constraint solving. For example, Kodkod already supports cost optimization problems (which we utilized in Chapter 3), and has incremental solving features. PBnJ can be updated to bring those features to the language level. An open question is whether or not local-search heuristics can be equally supported.

Chapter 5

5 Plan B: Falling Back on Executable Specifications

In the last chapter I introduced the PBnJ language and runtime system that lets us execute specifications declaratively at run time. We are finally ready to use it to support what I stated in my thesis.

I describe a new approach to employing specifications for software reliability. Rather than only using specifications to *validate* implementations, we can additionally employ specifications as a *reliable alternative* to those implementations. The approach, which I call Plan B, performs dynamic contract checking of methods. However, instead of halting the program upon a contract violation, I employ a constraint solver to automatically *execute* the specification in order to alter the program's run-time state so that it conforms to the specification and allow the program to continue properly. This chapter describes Plan B in the context of PBnJ (Plan B in Java) programs. I also describe our experience using the language to enhance the reliability and functionality of several existing Java applications.

5.1 Introduction

Many researchers have explored the use of *specifications*, for example pre- and postconditions on methods expressed in a variant of first-order logic, to gain confidence in the correctness of software. One approach employs specifications for static program verification, guaranteeing that each method meets its declared specification for all possible executions. In recent years this approach has been increasingly automatable via the use of constraint solvers (e.g., [FLL+02, BLS05a, DCJ06, ZKR08, ZKR09]). However, the limits of static verification make it difficult to scale this technology to complex programs and rich program properties. A complementary approach employs specifications for dynamic contract checking (e.g., [Mey97b, FF01]). In this style pre- and postconditions are checked as a program is executed. Performing the checking is straightforward since specifications are only enforced on a single run-time program state at a time. If a specification violation is found, there is little recourse other than halting the program. This is desired during development, as it gives the developer exact debugging information on what program locations have failed to conform to which properties. However, dynamic contract checking would not be useful in deployed software. It would be unacceptable to simply halt the program in many situations, when the software is already out of the hands of the developers.

In this chapter I explore a new approach to employ specifications for software reliability, which I call Plan B. The main idea is that specifications can be used not only to check an implementation's correctness but also as a *reliable alternative* to faulty or incomplete implementations. Like dynamic contract checking, my approach checks for violations of method postconditions at run time. However, rather than simply halting the program upon a violation, Plan B *falls back* on the specification itself, directly executing it in order to safely continue program execution. I observe that specifications can be executed using the same kinds of constraint solvers that are traditionally used for static verification. Rather than using the constraint solver to verify the correctness of a method for all possible executions, Plan B *ignores* the method implementation and lets the constraint solver search for a *model* that satisfies the method's postcondition given the dynamic program state on entry to the method.

Integrating executable specifications into a programming language in this fashion provides several benefits. As described above, Plan B can be used to safely recover from dynamic contract violations. Similarly, Plan B can safely recover from arbitrary errors that prematurely terminate a method's execution, for example a null pointer dereference or out-of-bounds array access. Finally, Plan B allows programmers to leverage executable specifications to simplify software development. For example, a programmer could implement the common cases of an algorithm efficiently but explicitly defer to the specification to handle the algorithm's complex but rare corner cases. While executing specifications can be significantly less efficient than executing an imperative implementation, current constraint-solving technology is acceptable in many situations, especially those for which the only safe alternative is to halt the program's execution. Furthermore, Plan B can take advantage of continual improvements in constraint-solving technology to broaden its scope of applicability over time.

There have been several recent research efforts on dynamic *repair* to recover from program errors (e.g., [DR03, DR05, EKVM07, EK08, NZGKM12]). These tools use heuristic local search to find a "nearby" state to the faulty one that satisfies an object's integrity constraints. The goal is to allow execution to continue acceptably, even in the face of possible data loss or corruption due to the fault. In the Plan B approach, however, we ignore the faulty program state, roll it back to the starting point, and "re-execute" the method using its postcondition (and the object invariants) by falling back on a general-purpose constraint solver. The goal of Plan B is not patching a corrupt state into an acceptable state, but rather correcting the *functionality* of a method should its implementation violate it. Therefore, Plan B is in general less efficient than repair, which leverages the fact that many errors only break invariants in local ways. However, Plan B is useful when it is important to fully recover from a fault rather than simply restoring the data resulting from it. Plan B also enables safe recovery from an arbitrarily broken program state and can ensure rich

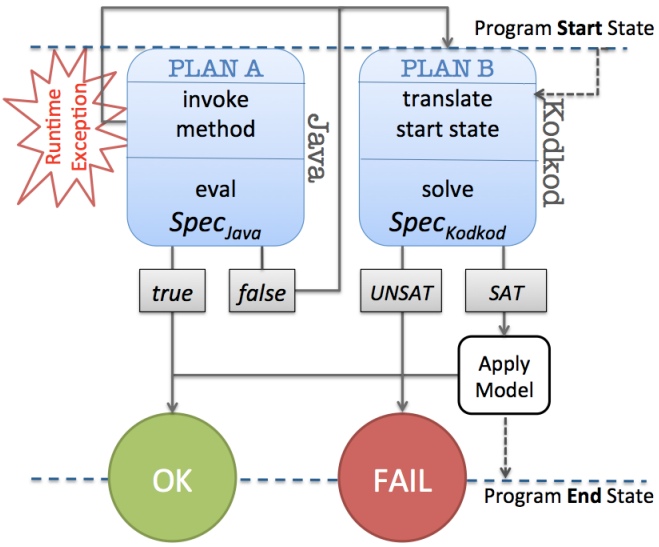


Figure 21: Method invocation in PBnJ for Plan B: falling back from Plan A (ordinary method execution) to Plan B (executing the method specification)

program properties that relate the input state of a method to its output state, both of which are challenging for local-search-based approaches.

After overviewing how I modify the method invocation strategy in PBnJ in order to support the online repair approach of Plan B, I describe my experience using PBnJ in several case studies (Sec. 5.4). First, I have written executable specifications for several common data structures. I use this case study as a stress test for my approach, employing complete specifications that ensure 100% correctness in the event of a fallback. Second, I show how executable specifications can enhance the reliability and functionality of several existing Java applications.

5.2 Using PBnJ for Plan B

5.2.1 Contract Checking and Recovery

Fig. 21 overviews how the original PBnJ design (see Fig. 18) was updated to enable declarative execution as a fallback mechanism for a Java method. The method is executed as usual. Upon normal completion, we check that the method obeys the declared object invariant and method postcondition by executing (Java translations of) these predicates. If the method invocation violates the declared specification, we fall back to the specification. Execution also falls back to the specification if the method terminates with a Java `RuntimeException` (e.g., an `ArrayIndexOutOfBoundsException` or `NullPointerException`). Fallback involves

translating the object invariant and method postcondition into the logic of the KODKOD relational constraint solver [Tor09] and invoking the solver to search for a *model* satisfying the specification. If a model is found we use the model to transform the current program state and continue the execution. If KODKOD reports unsatisfiability, then the specifications have no solution within the search bounds provided by the programmer (see Sec. 4.2.4). In such a case we throw a `ContractViolationException`, similar to what would happen with traditional contract checking.

If fallback is required, the approach requires a mechanism to roll the program state back to its starting point (on entry to this method). This is necessary to be able to recover from arbitrary errors in the implementation, e.g. one that entirely corrupts the starting data or crashes somewhere in the middle. To this end, before a method is executed, we make a deep copy of the reachable state from any object whose `old` field is mentioned in the method's postcondition [12]. This copy is then used as the starting state of the method when checking the postcondition, after its execution completes. If the postcondition is not satisfied, the preserved old state on entry to the method is translated into a set of relations that are provided to the KODKOD solver for the purpose of model finding.

This implementation choice incurs a performance overhead because of these deep copies, but this mechanism has allowed us to easily experiment with a variety of expressive postconditions. In the future I will explore several approaches to optimize the implementation of `old`. For example, research on program *replay* (see [CS98]) tells us how to recover a previous state in the execution by recording operations but without the need to perform any copying of data. It may also be reasonable in certain cases to switch to a more standard, shallow, notion of `old` as found for example in JML [LBR06].

Preconditions: As a feature only relevant to PLAN B, I also added support in PBNJ for user-defined preconditions on methods, in addition to postconditions. In cases when methods include preconditions, these are checked before the method body is invoked, and on a violation a `PreconditionViolationException` is thrown back to the caller. Note that this architecture is composable. For instance, if the caller itself has a postcondition, and crashes due to a `PreconditionViolationException` thrown by the callee, we know that the PLAN B fallback mechanism will be automatically triggered for the caller. Thus recovery by fallback also works in cases where a method is called with bad inputs.

[12] We make no attempt to roll back any data that is irrelevant to the postcondition. PLAN B guarantees the satisfiability of the user's functional specifications, but nothing more.

5.2.2 Two Usages for Fallback

I will next describe two possible ways the PLAN B system can be beneficial. It can be deployed for *accidental fallback*, to automatically recover from unforeseen failures due to errors that escape the validation process and creep into the shipped software. The approach complements the static and dynamic verification techniques, and employs functional specifications for recovering from software faults whenever they occur. It can be deployed in critical systems that are required to be robust in the face of occasional faults and crashes.

In the second mode, the developers may purposely omit parts of the code and use PBNJ to execute specifications declaratively by default. I call this usage *intentional fallback*, which is particularly useful as an alternative to computationally expensive tasks that are hard to code but simple to specify, such as search, scheduling, window layout, and so on. For example, a programmer could implement the common cases of an algorithm efficiently but explicitly defer to the specification to handle the algorithm's complex but rare corner cases.

Accidental Fallback: Let us go back to the linked list example in Fig. 17, which we described in terms of PBNJ specifications. Recall that we specified postconditions for the `sort` method without implementing it. Now, imagine that the `sort` routine is, in fact, implemented as illustrated by Fig. 22. (I renamed it `bubbleSort` here to be more descriptive.)

We can see how PLAN B helps ensure reliability in the face of program errors. The `bubbleSort` method implementation has two errors, which are marked in the figure. First, the guard in the `while` loop at marked line *A* should read `curr.next != last`. The error will cause the subsequent line to throw a `NullPointerException` when `curr.next` is `null`. Second, line *B* should read `prev = curr.next`. This error does not cause any exceptions to be thrown, but on some lists it will erroneously throw away elements. With the provided specification, PLAN B catches and successfully recovers from both errors by falling back to the external constraint solver. Aside from a slowdown in the application (depending on the size of the list being sorted), this recovery is completely transparent to the user and to the rest of the application.

Intentional Fallback: Fallback may be useful for a programmer to rely upon explicitly, in order to handle complex corner cases that arise in rare circumstances. SweetHome3D [Swe] is a popular interior design application implemented in Java. Users can add and arrange pieces of furniture in a room and view the results in a 3D view. The application has sometimes an odd behavior when we attempt to drag furniture pieces around: we may see pieces of furniture fused together in the 3D view. Trying to automatically

```
class List ensures isAcyclic()  {
    void bubbleSort() ensures this.isPermutationOf(this.old) && this.isSorted() {
        Node curr, tmp, prev = null, last = null;
        while (last != head) {
            curr = head;
            while (curr != last) {                    // A
                if (curr.value > curr.next.value) {
                    if (curr == head)
                        head = curr.next;
                    else
                        prev.next = curr.next;
                    tmp = curr.next.next;
                    curr.next.next = curr;
                    prev = curr;                      // B
                    curr.next = tmp;
                } else {
                    prev = curr;
                    curr = curr.next;
                }
            }
            last = curr;
        }
    }
}
```

Figure 22: The implementation and specification of the `bubbleSort` routine for the a linked list of integers. Full listing of specifications was shown in Fig. 17. The marked lines are discussed in text.

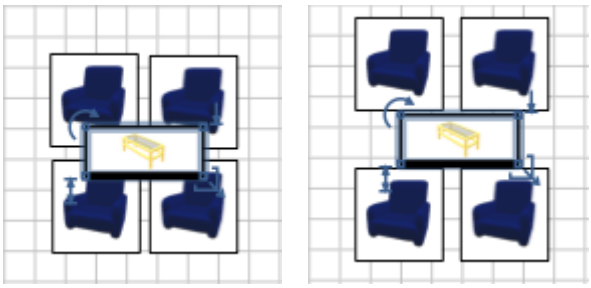
```
void moveFurnitureIfNeeded()
    ensures notOverlapped() && notTooFar() && keepRelativePosition();

spec boolean notOverlapped() {
    return all HomePieceOfFurniture p1 : this.furniture |
        all HomePieceOfFurniture p2 : this.furniture |
            (p1 == p2
            || (abs(p1.getX() - p2.getX()) >= ((p1.getWidth() + p2.getWidth())/2))
            || (abs(p1.getY() - p2.getY()) >= ((p1.getDepth() + p2.getDepth())/2)));
}

spec boolean notTooFar() {
    return all HomePieceOfFurniture p: this.furniture |
        ((abs(p.getX() - p.old.getX()) <= p.getWidth() / 2)
         && (abs(p.getY() - p.old.getY()) <= p.getDepth() / 2));
}

spec boolean keepRelativePosition() {
    return all HomePieceOfFurniture p1: this.furniture |
        all HomePieceOfFurniture p2: this.furniture |
            ((cmp(p1.getX(), p2.getX()) == cmp(p1.old.getX(),p2.old.getX()))
             && (cmp(p1.getY(), p2.getY()) == cmp(p1.old.getY(),p2.old.getY())));
}
```

Figure 23: Enhancing SweetHome3D to automatically rearrange overlapping pieces of furniture. The `getX` and `getY` methods return the coordinates of the center of a piece of furniture. The `cmp` method returns -1 if the first argument is less than the second argument, 0 if the arguments are equal, and 1 otherwise.



(a) (b)

Figure 24: (a) Four chairs and a coffee table overlapping the chairs (b) Fallback mechanism automatically rearranges the furniture.

rearrange shapes to avoid any physical overlaps is actually a complex problem, so the developers did not code in any feature to avoid overlaps.

Fig. 23 illustrates how PLAN B makes it easy to enhance the implementation of SweetHome3D to automatically rearrange furniture as pieces are moved or new pieces are added in a room to ensure that two pieces never overlap in space. This is a good application for intentional fallback, since it would be cumbersome and error prone to implement manually, and it may be reasonable to expect such rearranging to be necessary only rarely.

To implement the enhancement we simply augmented the existing method for adding or moving a piece of furniture to invoke the declarative method (empty body) `moveFurnitureIfNeeded` shown in Fig. 23. PLAN B's contract checking dynamically checks the method's postcondition and performs fallback if necessary, thereby allowing the programmer to safely ignore this special case. The declared postcondition ensures that there are no overlaps and that the new position of each piece of furniture is close to its old position and retains the same relative position to every other piece. Fig. 24 shows "before" and "after" screenshots for a simple example.

5.3 Implementation

Only few updates had to be done to the vanilla PBNJ compiler's instrumentation steps to enable it for PLAN B. As I noted previously, support for enforcing preconditions was added. Methods with a postcondition (or all public methods should object invariants be declared for the class) are instrumented as follows.

As mentioned, a deep-copying step of all data whose `old` versions are referred to in the specifications is done to ensure contract checking and fallback can be done regardless of any destructive updates by a faulty

method's body. We wrap the method body in a `try` block and use the `finally` clause to invoke the Java translations of the method's postcondition and any object invariants. We also use this `try` block to catch any run-time exceptions. If either contract checking fails or a run-time exception is thrown, we proceed to fall back to the KODKOD solver.

5.4 Case Studies

This section describes my experience using PBNJ specifications as a reliable fallback mechanism and evaluates the expressiveness and run-time performance of this approach. First I discuss the use of executable specifications to make common data structures like lists and trees robust to implementation errors. Then I describe my experience of providing a fallback mechanism for existing Java applications.

5.4.1 Fallback for Data Structures

Fig. 17 in Sec. 4.1 showed a portion of a linked list written in PBNJ. In addition to a complete linked list, I also implemented a binary search tree as well as a red-black tree. Fig. 25 shows a portion of our red-black tree. The object invariant ensures the various properties required of a red-black tree, which guarantee that the tree satisfies the usual binary search tree invariant and that the tree is balanced. The `nodes` specification method is similar to that from Fig. 17, but I use the `+` operator to take the union of the `left` and `right` relations. The `isBinarySearchTree` and `leaves` methods show how nested quantifiers and set comprehension provide a declarative and powerful mechanism for expressing complex invariants.

In order to evaluate the power and cost of our fallback mechanism, I have not provided any implementations of methods like `insert` and `delete`. Dynamically all invocations of these methods will fail to satisfy the declared postconditions, triggering a fallback to the specification. In this way, the program serves as a *self-describing, runnable interface* of a red-black tree, which can be used to ensure reliability for more efficient implementations. The code size is roughly five times smaller than a typical Java implementation of a red-black tree, mainly because of complex corner cases that imperative implementations of the insert and delete operations must handle.

The `insert` and `delete` operations include frame conditions which ensure that a node's value will remain unchanged in the face of a fallback. However, these conditions still allow the link structure of *every* node to be modified. Since the postconditions of these methods only ensure that the resulting tree has the correct

```
class RBTree ensures
    isBinarySearchTree() && rootBlack() && redsChildren() && eqBlacks() {

    class Node {
        Node left, right, parent;
        int value;
        boolean color;
        spec PBJSet<Node> descendants() { return this.^(left+right); }
    }

    Node root;

    spec PBJSet<Node> nodes() { return root.*(left+right); }

    spec PBJSet<Integer> nodeValues() { return nodes().>value; }

    spec PBJSet<Node> blackAncestors() {
        return { all Node n : this.*parent | n.color };
    }

    spec PBJSet<Node> leaves() {
        return { all Node n : nodes() | (n.left == null || n.right == null) };
    }

    spec boolean isBinarySearchTree() {
        return all Node n : nodes() |
            ((n.left == null
              || all Node lc : n.left.descendants() | lc.value < n.value)
             && (n.right == null
                 || all Node rc : n.right.descendants() | rc.value > n.value));
    }

    spec boolean eqBlacks() {
        return all Node l1 : leaves() |
            all Node l2: leaves() |
                (l1 == l2
                 || l1.blackAncestors().size() == l2.blackAncestors().size());
    }

    spec public boolean redsChildren() {
        return all Node n : nodes() |
            (n.color || all Node c : n.children() | c.color);
    }

    void insert(int value)
        modifies fields RBTree:root, Node:color, Node:left, Node:right,
            Node:parent
        adds 1 Node
        ensures nodeValues().equals(old.nodeValues().plus(value));

    void delete(int value)
        modifies fields RBTree:root, Node:color, Node:left, Node:right,
            Node:parent
        ensures nodeValues().equals(old.nodeValues().minus(value));
}
```

Figure 25: A portion of our red-black tree with executable specifications in PBnJ

Table 8: Fallback pre- and post-processing overhead, including copying, contract checking, and conversion to KODKOD (*fb.*), KODKOD's translation to SAT (*tr.*) and SAT solving time (sec.) (*sat.*) using Glucose [AS09] of a fallback event on an `insert` call in a binary-search tree (BST) or red-black tree (RBT) and a `bubbleSort` call on a linked list (List), with n nodes. I report timings without object frame conditions (*no frame*) and with them (*with frame*). Solving on a Core i7-3930K, 3.20GHz, with 8-bit integers. Timeout $t/o = 600$.

	BST						RBT						List		
	insert												bubbleSort		
Size	**no frame**			**with frame**			**no frame**			**with frame**			**no frame**		
(n)	*fb*	*tr*	*sat*	*fb*	*tr*	*sat*	*fb*	*tr*	*sat*	*fb*	*tr*	*sat*	*fb*	*tr*	*sat*
10	.05	.82	.09	.05	.77	0	.05	.90	1.7	.05	.84	0	.05	.47	.01
20	.05	1.1	50	.05	1.3	0	t/o			.05	1.7	0	.05	.96	.20
40	t/o			.1	3.9	0.1	t/o			.1	6.9	.21	.05	2.9	8.2
60	t/o			.1	11	0.3	t/o			.1	22	.66	.1	8.5	53
80	t/o			.1	31	1.0	t/o			.1	67	1.9	.2	25	187
100	t/o			.1	71	1.9	t/o			.1	148	3.8	.2	57	495

values, fallback may alter the tree in a manner that differs from a typical implementation, but the resulting tree will still satisfy the red-black tree invariants and contain the proper values.

One way to preserve the structure of the original tree upon an `insert` or `delete` operation is to include a `modifies objects` clause. Doing so is fairly straightforward for a binary search tree. For example, there is always only a single node in the tree that is affected by an insertion operation. Therefore the programmer can include a clause on `insert` as follows, which invokes a function that produces the singleton set containing the affected node:

```
modifies objects getParentToBeFor(value)
```

The same thing can be done for the red-black tree, but in that case computing the set of nodes affected by an insertion or deletion is more complex due to the need to potentially rebalance the tree.

Performance: I employed the data structures described above as a stress test for our fallback mechanism, using fallback to guarantee complex invariants with 100% functional recovery from an arbitrary failure. Table 8 shows the running times of a fallback event for an insertion into a binary search tree, an insertion into a red-black tree, and an invocation of `bubbleSort` for the linked list from Fig. 22, for various sizes of the data structures. Without object frame conditions the KODKOD-based fallback mechanism is only practical for relatively small trees. However, when object frame conditions are provided the approach becomes feasible up to a 100-node tree. The object frame conditions keep the number of unknowns roughly the same as the problem size increases, so SAT solving time (*sat*) scales well. The main bottleneck is instead

KODKOD's translation from a relational logic formula to a SAT formula (tr). In the future I would like to explore techniques for optimizing this encoding step, for instance employing an encoding optimization done in SQUANDER [MRYJ11].

The main reason KODKOD is inefficient on these constraints is the presence of free integer variables representing the `value` field for the `Node` objects, despite the fact that we only assume 8-bit integers. SAT-based constraint solvers in general are not optimized for handling numeric constraints.

Comparison with Data Structure Repair Techniques: The PLAN B approach is complementary to that of recent online data repair tools. Such tools are very efficient and useful when data is broken in local ways and some data loss or corruption is acceptable. This approach is more expensive but can recover the intended semantics of a faulty method and can properly recover from arbitrarily broken program states. To concretely illustrate these differences, I ran the Juzi repair tool [EK08] on our binary search tree using intentionally broken implementations.

First I modified the `insert` method of the BST implementation to corrupt a single node and asked Juzi to restore the binary search tree invariant but not the postcondition of `insert`. This kind of local repair is ideally suited for Juzi, which satisfies the binary search tree invariant in 0.1 seconds for a tree with 10 nodes. In contrast, PBNJ reverts to the state before the faulty method was invoked, so it cannot leverage the locality of the error. Aside from increasing the cost of repair, this choice means that without including the method postcondition PLAN B is likely to produce a trivial solution such as an empty tree. I then augmented the tree's object invariant to include the postcondition for `insert` by manually maintaining a field denoting the original set of nodes on entry to the method. In that case Juzi timed out after a minute. On the other hand, PBNJ recovers from the corruption and additionally ensures that the insertion happens properly in a second without object frame conditions and 0.3 seconds with them.

5.4.2 Fallback for Existing Java Applications

I ported several existing Java applications to PBNJ, allowing me to explore the expressiveness of PBNJ's specification language as well as the practicality of fallback for various kinds of constraints. In addition to the SweetHome3D application described in Sec. 5.2, I ported Java's `GridBagLayout` class and an open-source implementation of chess. Since these applications rely on the collection classes in Java's `java.util` library, I also compiled PBNJ versions of many of those classes (e.g., `ArrayList`). This entailed turning some existing methods into `spec` methods (e.g., `size()`) so they could be used in clients' specifications and adding new

```
spec boolean arrangeGridLayoutValid() {
    // for any given component in the window:
    return all Component c1 : components |
        (boundsValid(c1) && sizeValid(c1) && positionValid(c1) &&
         all Component c2 : components |
             (c1 == c2  || (noOverlaps(c1,c2) && relPositionsValid(c1,c2))));
  }
```

Figure 26: My specification for the `arrangeGrid` method in `GridBagLayout`

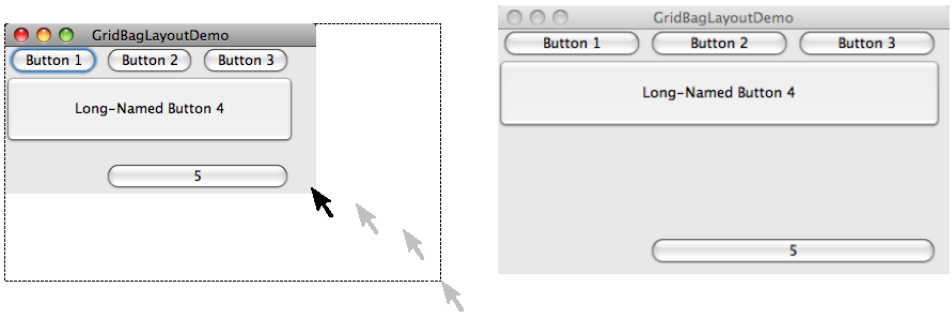
`spec` methods as necessary. In order to support quantifying over a collection, I also implemented a `toPBJSet` specification method for each collection class, which returns a set of the collection's elements.

GridBagLayout: The layout task in GUI applications is often complex. While individual constraints are usually simple arithmetic restrictions, laying out a window with many different components with both individual constraints and dependencies among one another is non-trivial. The `java.awt.GridBagLayout` class from Java's widely used Abstract Window Toolkit (AWT) library is a case in point[13]. This class is perhaps the most flexible of Java's layout managers, allowing components of varying sizes to be laid out subject to a variety of constraints.

I augmented several methods in `GridBagLayout` with PBnJ specifications. The main layout (and most involved) method in `GridBagLayout` is `arrangeGrid`, which is invoked whenever a user makes any change to the window (e.g., resizing) and contains over 300 lines of code. I used the informal documentation provided by Java to provide a partial specification for this method. My specification, which is shown in Fig. 26, requires that each component is located within the bounds of the window, is resized appropriately with respect to the window size, satisfies various position constraints (e.g., each row in the grid is left- and right-justified), does not overlap any other component, and retains its position relative to other components. PBnJ supports quantification over arrays (such as the `components` field in the figure) in addition to `PBJSet`s. The complete specification including helper methods is 35 lines of code.

To execute its specification, I removed the original body of `arrangeGrid` so that fallback would occur on each invocation. Fig. 27 shows a screenshot of the initial layout for a window with five buttons using my specification, as well as the layout after the user resizes the window. The fallback mechanism takes around 5 seconds on average in each case, and the result is indistinguishable from that of the original `arrangeGrid` implementation. Although not yet an acceptable performance overhead to use as a complete replacement for the original implementation, PBnJ provides a practical way to ensure reliability of an implementation in the face of a crash or incorrect layout.

[13] See `http://www.youtube.com/watch?v=UuLaxbFKAcc` for a funny video about the difficulties of using `GridBagLayout`.



(a) (b)

Figure 27: Using executable specification for `arrangeGrid` (a) to layout a window originally and (b) after a resize event

I also experimented with a simple optimization that cuts down the time significantly with little impact on the results. The `arrangeGrid` method uses constraints that are described in terms of the entire screen's pixel coordinates (e.g., 1024x768), which constitutes a large search space. I experimented with a version of my specification in which I solve a scaled version of the problem by dividing all coordinates by a fixed constant (10 in my experiment), solve for a model, and then multiply the constant back to get the actual coordinate values to use. This approach reduced fallback time to under a second with little perceptible difference in the resulting layout. The fallback time for the SweetHome3D application shown in Fig. 24 was also reduced to less than a second after incorporating the same optimization technique. In general, PBnJ allows programmers to provide such underspecified postconditions, which can be used to allow practical fallback at the expense of some degradation in the quality of the result.

JChessBoard: JChessBoard [JCh] is an open-source chess implementation in Java. The application has a feature to highlight the valid moves for a piece clicked by the user. The same method is also called when generating candidate moves for the computer player. For this case study, I annotated that method, `getPossibleMoves`, with a PBnJ postcondition that itself computes the set of valid moves and compares the result to the result of `getPossibleMoves`'s implementation. This case study demonstrates the ability of our specification language to express complex properties and perform sophisticated computations.

Fig. 28 shows the specification method that computes the valid moves for a given piece on the board. JChessBoard uses a single-dimensional array `board` of size 64 to represent the board, and a given square (x, y) is indexed using the formula $8x + y$, where x and y are in the range [0,7]. In order to generate the set of valid moves, JChessBoard iterates over a statically calculated Java `Vector` named `allMoves` — the collection of all moves that can possibly originate from each square, assuming the square could hold any

```
spec PBJSet<Move> allValidMovesFrom(Piece p, int from) {
    return { all Move move : allMoves.toPBJSet() |
        (move.getFrom() == from && isValidMove(p, from, move.getTo())) };
}

spec boolean isValidMove(Piece p, int from, int to) {
    if (p == BISHOP)
        return isValidBishopMove(from, to);
    else if ...
}
```

Figure 28: Computing valid chess moves as a PBnJ specification

possible piece. My specification quantifies over this vector in order to obtain only moves for the given square `from` that are valid for the given piece `p`.

Fig. 29 shows the specification method `isValidBishopMove`. The method checks that the `from` and `to` squares are on the same diagonal and that all intervening squares on that diagonal are empty. This specification makes use of many of the features of our specification language, including assignments to local variables, nested quantification, integer interval ranges for quantification, array accesses, and bitwise operations on integers. Except for the quantification expressions, the rest of the code comes from the original JChessBoard implementation; I simply added `spec` annotations. Nonetheless, this code can be automatically translated to Kodkod for the purposes of automatic fallback.

A fallback event for `getPossibleMoves`, which entails "executing" the above method `allValidMovesFrom` in Kodkod, takes 2-3 seconds on average. I have also explored an optimization which replaces the `allMoves` field with a separate statically computed vector per type of piece (`allBishopMoves`, `allQueenMoves`, etc.). Using the appropriate vector for the given piece during a fallback event rather than the generic `allMoves` vector reduces the execution time to about 0.5 seconds.

5.5 Related Work

This work builds upon several lines of research on software reliability.

5.5.1 Data Structure Repair and Self-Healing Systems

My work is inspired in part by recent work on online repair of data structures. As mentioned in earlier comparisons, I view the approaches as useful in different scenarios. Repair is useful when it is important

```
spec boolean isValidBishopMove(int from, int to) {
    int fromRow = getRow(from), fromCol = getCol(from);
    int toRow = getRow(to), toCol = getCol(to);
    return Math.abs(toRow - fromRow) == Math.abs(toCol - fromCol)
        && checkDiagonalLineOfSight(fromRow, fromCol, toRow, toCol);
}

spec boolean checkDiagonalLineOfSight
    (int fromRow, int fromCol, int toRow, int toCol) {
    int minRow = Math.min(fromRow, toRow);
    int maxRow = Math.max(fromRow, toRow);
    int minCol = Math.min(fromCol, toCol);
    int maxCol = Math.max(fromCol, toCol);
    return all int r : minRow + 1 .. maxRow - 1 |
        all int f : minCol + 1 .. maxCol - 1 |
            (Math.abs(r - fromRow) != Math.abs(f - fromCol)
            || board[getSquare(r,f)] == EMPTY);
}

spec int getRow(int square) { return square >> 3; }

spec int getCol(int square) { return square & 7; }

spec int getSquare(int row, int column) { return (row << 3) + column; }
```

Figure 29: Specifying the valid bishop moves from a square

for an application to continue executing despite the presence of errors, while Plan B is useful when it is important for an application to achieve the intended functionality of a faulty method before continuing.

A survey of recent work on automated online repair can be found in [Rin12]. The repair approach of Demsky and Rinard [DR03, DR05] allows programmers to provide high-level specifications in terms of an abstract model of objects, along with a translation from the concrete to the abstract worlds. These specifications are checked dynamically and violations invoke a repair algorithm that employs a specialized set of repair actions (e.g., add or remove an element from a set). The authors provide several case studies illustrating the practicality of the approach for surviving errors in a variety of existing applications. The use of abstract models allows for a high-level approach to repair but also places additional burdens on programmers. It may be useful for us to consider incorporating user-defined abstractions in PBnJ, which could allow for higher-level encodings into Kodkod that are more efficient than my current concrete encoding of a program state.

Elkarablieh *et al.* describe *assertion-based* repair of data structures [EGSK07, EKVM07, EK08]. Their approach takes the broken program state along with a Java method representing the violated assertion predicate and performs a heuristically guided and bounded state-space exploration to search for a nearby state that satisfies the predicate. This approach is in some ways analogous to my use of SAT-solving

technology to perform a bounded search. However, my search begins from the *pre-state* of the faulty method and strives to achieve the intended functionality of the method, while their search begins from the *post-state* of the faulty method and strives to perform local repairs to satisfy the class's integrity constraints. My experiments in Sec. 5.4 compared against the Juzi tool directly. I have not conducted an evaluation of a couple of more recent tools (e.g. [NZGKM12]) that have since followed that work, however I believe the goals of data structure repair and Plan B remain separated. These repair tools employ symbolic execution [Kin76] along with automatic decision procedures to solve for the values of integers and other primitives and employ static analysis to optimize the repair. Both ideas would be useful in our context as well in order to speed up fallback.

Carzaniga *et al.* [CGP08] suggest that the inherent redundancies existing in large software systems can act as *workaround* (which I call *fallback* here) for failing components. For example, the system may realize there are more than one series of invocations that can achieve a certain operation. Using this knowledge a self-healing system can try all known workarounds when the first attempt results in a failure. Plan B is similar to such a system, except that it relies on constraint solving as the workaround.

5.5.2 Declarative Execution

Plan B can be seen as a special case of a mixed interpreter, where constraint solving is used only as a fallback mechanism rather than as a "first-class" part of the language. I believe that fallback is a compelling use of declarative execution that may be more practical to support than a full-fledged mixed interpreter. For example, it is reasonable for PBnJ to employ bounded constraint solving. Although this may result in missed opportunities to recover a program, in the worst case we can simply throw an exception as traditional contract checking would do. In contrast, it would be unreasonable for a general mixed interpreter to sometimes fail to execute a specification that is used as part of a program's implementation.

5.6 Discussion and Future Work

The PBnJ implementation can be optimized to reduce the overhead of fallback. I plan to explore ways to avoid the deep copying that we currently perform due to the uses of the `old` field. One possibility is to use static analysis to determine the parts of an object's state that cannot change and therefore need not be copied. Another option is to use a copy-on-write strategy, where state is only copied just before it is overwritten. Researchers have worked out many forms of program *replay* (e.g. [CS98]), which can recover a previous state in the execution by recording operations but without the need to perform any copying of

data. In the future I would like to investigate deploying replay for PLAN B. It may also be reasonable in certain cases to switch to a more standard, shallow, notion of `old` as found for example in JML [LBR06]. Finally, with more experience we may find that an alternative semantics for `old` is more practical while still providing the desired expressiveness.

An interesting question to investigate is whether the results of automatic fallback can be used to help developers localize and correct errors in their implementations.

5.7 Conclusion

I have presented the PLAN B approach to software reliability. The main contribution of this chapter is the notion that formal specifications, when made *executable* by means of a SAT-based constraint solver, can act as reliable alternatives for incomplete or faulty method implementations. As a secondary benefit, such specifications can also be used directly to make implementations more declarative and reliable by construction. I have demonstrated both use cases via example in the PBNJ extension to Java and presented our experience using the language to guarantee rich properties on existing Java applications. I am excited about the possibilities of leveraging modern constraint solving technology as an online tool for software reliability. While many challenges remain, I am encouraged by our initial results and believe that there are several fruitful avenues for future research.

Chapter 6

6 Declarative Mocking: Executable Specifications as Mock Objects

In Chapter 4 I introduced the PBnJ declarative execution tool, and in Chapter 5 I demonstrated its application for building on specification-based validation to enable automated online repair. We have seen that Plan B may be used in *intentional* mode, that is, when the developer decides to omit all or certain parts of an implementation by letting the runtime declarative-execution fallback system handle a (complex) computation.

In this last chapter, I will describe my final instantiation of the idea of this thesis. Focusing again on unit testing, specifically this time on test-driven and agile development methodologies, we will apply intentional fallback to enable the execution of software components which are unavailable, or too cumbersome to fully include and/or set up, during testing.

Test-driven methodologies encourage testing early and often. *Mock objects* support this approach by allowing a component to be tested before all depended-upon components are available. Today mock objects typically reflect little to none of an object's intended functionality, which makes it difficult and error-prone for developers to test rich properties of their code. Here I present *declarative mocking*, which enables the creation of expressive and reliable mock objects with relatively little effort. In this approach, developers write method specifications in a high-level logical language for the API being mocked, and a constraint solver dynamically executes these specifications when the methods are invoked. In addition to mocking functionality, this approach seamlessly allows data and other aspects of the environment to be easily mocked. We will again deploy PBnJ for this purpose.

I have performed an exploratory study of declarative mocking on several existing Java applications, in order to understand the power of the approach and to categorize its potential benefits and limitations. I also performed an experiment to port the unit tests of several open-source applications from a widely used mocking library to PBnJ. I found that more than half of these unit tests can be enhanced, in terms of the strength of properties and coverage, by exploiting executable specifications, with relatively little additional developer effort.

```
414 class MySet implements Set {
415     List elems;
416     void add(Object o) {
417         if (!elems.contains(o))
418             elems.add(o);
419     }
420     void testAdd() {
421         List mockList = mock(List.class);
422         MySet s = new MySet(mockList);
423         when(mockList.contains(0)).thenReturn(false);
424         s.add(0);
425         verify(mockList, times(1)).add(0);
426         when(mockList.contains(0)).thenReturn(true);
427         s.add(0);
428         // shouldn't add duplicates:
429         verify(mockList, times(1)).add(0);
430     }
431 }
```

Figure 30: Using the Mockito mocking library to test an implementation of sets against a mock list object

6.1 Introduction

> "*Mock Objects ... tie tightly the test code to the implementation code.*"
>
> —"Mocks Suck (and what to do about it)", Talk by Brian Swan [Swa]

In Chapter 2, Sec. 2.1.4 we read a background on *mock objects*. The creators of mock objects emphasize that these are more than simple stubs [Fow], and enable *behavior*-verification of software-under-test. For example, the user can track what calls have been made on the mock, so to indirectly test whether the code interacting with the mocked components behaves as expected.

6.1.1 Motivating Example

As a small example, Fig. 30 shows how Mockito, a mocking library from Google, can be used to test an implementation of `Set` interface that is internally built using a list object that, hypothetically, is not yet available. The programmer's goal is to test that the `add` method for `MySet` avoids adding duplicate elements. Unfortunately, this seemingly simple task is not particularly straightforward. The call to `mock` on line 421 creates a mock object that meets the `List` interface. By default this mock does not faithfully implement the intended behavior of `List`'s `contains` and `add` methods. For example, the mock has no way of knowing what boolean value to return upon a call to `contains`, so the programmer is required to explicitly provide this information. Therefore, on line 423 the programmer indicates that `contains` should return `false` when

given the argument value 0. Worse, she has to update this information on line 426, since 0 has now been (conceptually at least) added to the list. Similarly, because the mock list's `add` method does not actually add the given element to the list, the programmer cannot directly check that duplicates are handled properly. Instead, she uses Mockito's `verify` method to check the number of times that the mock object's `add` method is invoked (lines 425 and 429). These checks ensure that the mock object's `add` method is not invoked on the second invocation of `s.add(0)`, which implies that the duplicate element is properly ignored.

Mock objects in the form they exist today are undoubtedly useful as they enable unit tests to be performed despite missing dependencies, with very little effort. Yet they are severely limited in the benefits they can provide to programmers. Typically a mock object is just a stub, with little if any of the actual functionality of the code it is mocking. Therefore, mocking libraries (e.g. Mockrunner [AI], Mockito [Fab]) require clients to explicitly indicate the results they expect from the mock object and to only indirectly test the correctness of their code through implementation-specific checks. As a result, tests are fragile, error-prone, and difficult to understand or reuse.

The limitations of mock objects are clear on the simple example we saw in Fig. 30, and these problems are exacerbated as the objects being mocked and the code being tested become more complex. To overcome these limitations, my goal is to enable programmers to easily build mock objects that directly reflect important parts of the functionality that they mock. Another goal, related to Brian Swan's quote I included in the beginning of this section, is to have mocks that are less coupled with any specific implementation of both the code under test and the code being mocked. Of course, any practical approach should require much less effort than it would take to actually implement the object being mocked, or else the benefits of mocking are lost.

6.1.2 Declarative Mocking

I observe that recent progress on executable specifications and declarative execution can naturally support our goal: programmers can write specifications in a high-level logical language for the methods in the API being mocked, and a constraint solver dynamically *executes* these specifications when the methods are invoked. Specifications are often simpler than imperative code because they can directly express *what* behavior is desired without specifying *how* that behavior is achieved. Specifications also naturally support nondeterminism, which is useful both for modeling actual nondeterminism in the mocked object (e.g., the order in which messages will be received over the network) and for enabling *partial* specifications, with the nondeterminism used to represent "don't care" situations. I call the resulting approach *declarative mocking*.

Note that the effort in writing specifications can be amortized over their many benefits: while stubs like those in Fig. 30 must be carefully tailored to each individual test, specifications can define the behavior of a mocked API once and for all. Furthermore, the same specifications can be employed for static verification and dynamic contract checking of the "real" component being mocked.

In addition to using executable specifications to mock functionality, I observe that the same technology naturally supports the mocking of *data*, which provides additional value in the context of program testing. Unit tests commonly require a precondition to be established, i.e. the inputs and state of a program need to be initialized to satisfy some relevant properties, before the test can be executed. Executable specifications remove the need for imperative code to perform this initialization, instead allowing the tester to directly specify the intended precondition, reducing tester effort and increasing understandability.

Several recent works have used constraint solving to generate mock objects as part of an approach to automated test-case generation [GMW10, TS06, KTH09, MXT+09, ZML+12]. Declarative mocking uses similar technology but for a different purpose. Whereas the prior works employ a constraint solver offline in order to generate high path-coverage tests, declarative mocking employs a constraint solver online to dynamically substitute for missing code or data. Therefore, declarative mocking still requires users to provide their own tests. On the other hand, declarative mocks are fully executable with arbitrary inputs, independent of any test cases. For example, these mocks can be used to perform system-level testing without having to generate explicit system-level tests, and they allow users to easily interact with a running system, where mocks fill in for any missing functionality, to manually test features of interest.

6.1.3 Implementation and Evaluation

I added several features to PBnJ to support declarative mocking more naturally and evaluated declarative mocking with PBnJ in two ways. First, I performed an exploratory study on four open-source applications that I considered good targets for declarative mocking. These applications respectively interact with a web server, a database, a server that implements a file-transfer protocol, and a data center providing computational resources in the cloud. I used this study to classify various scenarios under which declarative mocking can potentially provide value over traditional mocks, and I also identify potential limitations of the approach.

Second, I ported six existing applications from Google Code that use Mockito to instead use PBnJ and I analyze the results using the classification derived from the exploratory study. Concretely, this second study is designed to answer the following two research questions:

RQ1 What is the overhead for a developer to use declarative mocks, when used to replicate the exact behavior of traditional mocking approaches today?

RQ2 How often and under what scenarios do declarative mocks offer advantages beyond the traditional approaches, with a justifiable amount of additional effort?

I investigated RQ1 by first porting each existing Mockito-based unit test to use PBnJ in such a way that the exact behavior is preserved. This experiment is something of a "worst case" for PBnJ, since specifications are used in a very limited and unnatural manner. I observed that on average twice as much developer effort (estimated by the number of lines of code) is needed to employ specifications instead of stubs. Further, there was an average added delay of one second per test in execution times.

To investigate RQ2, I revisited the same benchmarks to see which unit tests can be enhanced, with reasonable additional effort, by taking advantage of the benefits of executable specifications that were identified during the exploratory study. The enhancement is measured in terms of increased test coverage, code reuse, and/or strength of properties tested. According to this metric, 54% of the unit tests in the applications can be enhanced by employing executable specifications. For the rest of the tests, stubs are sufficient and any additional effort to enhance the mocks with specifications was not justified. Finally, since the specifications for a mocked API are generally reusable across an entire test suite, I observed that the ported tests have on average the same number of lines of code as the original unit tests.

This final chapter is organized as follows. I introduce the idea of declarative mocking in Sec. 6.2. Sec. 6.3 and 6.4 respectively present my exploratory study and my experiment with Mockito-based tests from Google Code. Finally I discuss related work and conclude.

6.2 Declarative Mocking

Traditional mocking paradigms are inexpressive and do not allow reuse. Declarative mocking through executable specifications allows the developer to avoid having to specify the concrete outcome of every interaction with a mock object, and instead directly and declaratively express the mock object's important behaviors. The use of executable specifications additionally enables a new form of mocking, in which the data and/or environment needed for a test case is produced via specifications. In this section I describe declarative mocking of both functionality and data and also introduce extensions to PBnJ to support these tasks.

```
class MockList implements List {

    Object[] elems, int size;

    spec int size() { return size; }

    pure boolean contains(Object o)
        ensures result <==> some int i : 0 .. size - 1 | elems[i].equals(o);

    void add(Object o)
        modifies fields MockList:elems, MockList:size
        ensures size == this.old.size + 1
            && elems[this.old.size] == o
            && all int i : 0 .. this.old.size - 1 |
                elems[i] == this.old.elems[i];
}
```

Figure 31: A runnable mock `List` class in PBnJ

6.2.1 Mocking Functionality

Recall the example in Fig. 30 where we used mocking to test that the `add` method for our `MySet` class avoids adding duplicate elements. We used Mockito to create a mock object from the `List` interface, stub the behavior of the mock's `contains` and `add` routines, and verify that the latter method is not invoked twice on the same element.

Figs. 31 and 32 illustrate how executable specifications resolve the problems for mocking that we saw in Fig. 30, without much extra effort. The `MockList` class shown in Fig. 31 uses PBnJ specifications to describe the intended semantics of its `contains` and `add` methods. The `MockList` specifications naturally capture the expected behavior of the two list operations. Furthermore, with PBnJ the result is a fully executable list implementation.

Given this declarative mock `MockList`, we can see in Fig. 31 how we can go about testing our `Set` implementation in the usual way. Because the mock object using executable specifications already "knows" the intended behavior of a list's operations, there is no need for each individual test case to specify this information. Similarly, test cases can use ordinary `assert`s to directly ensure properties of interest, rather than the indirect and fragile approach to checking behaviors in terms of method invocation counts. I will classify and quantify the benefits and limitations of declarative mocking versus traditional mock objects in the experimental studies of Sec. 6.3 and 6.4.

```
class MySet implements Set {

    List elems;

    int size() { return elems.size(); }

    void add(Object o) {
        if (!elems.contains(o))
            elems.add(o);
    }

    void testAdd() {
        List mockList = new MockList();
        MySet s = new MySet(mockList);
        s.add(0);
        s.add(0);
        // shouldn't add duplicates:
        assert (s.size() == 1);
    }
}
```

Figure 32: Testing a `Set` implementation using the PBnJ `MockList` class from Fig. 31

6.2.2 Mocking Data and Environment

Developers commonly want to test a feature in their application under various scenarios. For each scenario, they write initialization code to build up the state to the appropriate condition and then perform the tests. I observe that executable specifications can be naturally used to automatically modify program state to satisfy specified initialization conditions, relieving the tester of this burden. I call this approach *data mocking*, which is useful even for tests that do not rely on our declarative mock objects.

The `assume` Statement: To enable data mocking in PBnJ, I introduce a new statement of the form `assume` <*pred*>, where *pred* is a predicate on the current program state. When such a statement is encountered, PBnJ uses Kodkod to identify a program state satisfying *pred*, updates the program state, and continues execution. In the context of testing, the `assume` statement is useful both to synthesize the inputs to use in the test as well as to properly set the state of the mocked objects. I refer to the latter capability as *environment mocking*.

Consider again the `MySet` class with a mocked `List` object. In Fig. 33 I use environment mocking to easily set up two different test scenarios of interest: when the mocked list is non-empty and when it contains the `null` value as an element. When each test is run, PBnJ will nondeterministically find a state of the mocked object satisfying the given initialization condition.

```
class MySet {

    List elems;

    void test1() {
        assume elems.size() > 0;
        // now run the test ...
    }

    void test2() {
        assume elems.contains(null);
        // now run the test ...
    }
}
```

Figure 33: Mocking the environment for test initialization

The `unique` Modifier: In order to allow the developer to take full advantage of nondeterministic specifications and achieve higher coverage, I introduce an annotation for specifications called *unique*. This annotation can appear as a modifier on a method as well as on an `assume` statement. Consequently, every invocation of the associated specification on the same inputs will choose a result not previously chosen, unless all possible solutions have already been produced. For example, when the postcondition `result * result == 9` is invoked for the first time, either solution `result = 3` or `result = -3` may be produced. Now, should the `unique` modifier be present, a second invocation (within the same process) will only return the solution not previously chosen. In this way, a tester can cycle through all possible scenarios satisfying a given initialization condition. I implemented this feature by leveraging the Kodkod solver's ability to (surprisingly efficiently [14]) solve for all possible solutions within a given set of bounds.

6.3 Exploratory Study

This section reports on an exploratory study I performed in order to gain insights into the benefits and limitations of declarative mocking. I used PBnJ specifications to perform mocking on four open-source applications. This process allowed me to freely experiment without being bound to how mock objects are used today. Below I present the experiments and classify the advantages and disadvantages of declarative mocking that I discovered.

6.3.1 Applications

Let me start by a brief description of the applications that were part of this experiment.

[14] Kodkod is optimized to take advantage of possible isomorphism/symmetry in models [Tor09].

JStock—Mocking Webserver Data: JStock [jst] is an open-source Java stock watchlist GUI application, which frequently queries a web page to display live quotes in a table. I augmented JStock's source code with executable specifications to mock data that is received from the webserver, allowing me to test the application without accessing the network or requiring any web service libraries.

JDBC—Mocking Database Behavior and Data: The Java Database Connectivity (JDBC) API [JDB] allows Java applications to interact with database management systems using simple method calls with strings of SQL statements as their parameters. I used PBnJ to create a functional, in-memory mock database meeting this API.

TFTP—Mocking Errors and Network Nondeterminism in a Client-Server Protocol: TFTP [Sol92] is a simple protocol for transferring files between a client and a server. I created a mock server object in order to test an implementation of the client. Writing an imperative mock server that responds appropriately to client messages is relatively straightforward. However, by running the mock server locally, we lose the nondeterministic behavior that may occur due to dropped or misordered packets over the network. My goal was to explore the use of specifications to express this network nondeterminism in a declarative manner.

Hadoop—Mocking the Cloud's Behavior and Environment: Hadoop [had] is an open-source framework for processing MapReduce jobs in the cloud. Testing a MapReduce application is a challenging task. Using cloud resources is typically not a practical option for development and testing. Moreover, the performance of a MapReduce job is greatly influenced by numerous execution factors. These include the resources dedicated to the job, the workload, as well as scheduling policies.

MapReduce simulators have been built (e.g. [Tan]) to help developers simulate their applications locally. To utilize the simulators, the users are required to provide cluster and workload trace information from real previous executions on the cloud. This data is not always available and tedious to produce synthetically [WBMG11]. My goal was to use data mocking to produce realistic trace information for input to such simulators. Secondly, I employed specifications to mock Hadoop's standard FIFO and fair (HFS) schedulers (see [Jon]), which these simulators rely upon. The intention here was to experiment with the usage of specifications for rapid prototyping and design experimentation.

```
class Table ensures uniqueRows() {

    String primaryKey;

    List<String> columns, List<Tuple> rows;

    spec boolean uniqueRows() {
        int primaryIdx = columns.indexOf(primaryKey);
        return all int i : 0 .. rows.size() - 1 |
            all int j : 0 .. rows.size() - 1 |
                (i != j ==> rows.get(i).get(primaryIdx) !=
                    rows.get(j).get(primaryIdx));
    }
}

class Tuple extends ArrayList<Literal> { }
```

Figure 34: Partial invariants of a JDBC `Table` object

6.3.2 Advantages and Disadvantages

I now present the results of this exploratory study in terms of a classification of the advantages and disadvantages of declarative mocking. On each point, I compare my proposed approach with both traditional stub-based mocking and mocking by simply writing imperative code.

Advantages: Below is a list of properties that I found useful in declarative mocks.

Data Integrity – Objects often come with implicit integrity constraints that should be always satisfied. One of the benefits of declarative mocking is that these integrity constraints can be stated once and for all (as object invariants), and the runtime guarantees that any mocked behavior or data always conforms to these constraints. On the other hand, if done manually, it is easy for the tester to accidentally set up a program state that does not in fact conform to the necessary integrity constraints.

Example1. The in-memory JDBC-style database mock leverages the ability to express integrity constraints. Fig. 34 shows the representation of a database `Table` in the mock `Jdbc` objects, with `Literal` representing literal values storable in a cell. Each `Table` object must satisfy the `uniqueRows()` specification to enforce the exclusion of rows with duplicate primary keys. Consequently, any mocked `Jdbc` database or operation automatically ensures this property is preserved.

Declarative Expression – The expressiveness of specifications depends upon the flexibility of the employed solver. In performing these experiments, I found PBnJ's specification syntax to be adequate for concise and declarative expression of user intentions.

```
class DatabaseGUI {

    Jdbc jdbc, ResultSet results;

    void buttonTest1() {
        assume databaseInit() && results.size() == 0;
        // now test button behavior
    }

    void buttonTest2() {
        assume databaseInit() && results.size() == 2;
        // now test button behavior
    }

    spec boolean databaseInit() {
        String dbID = "shop", tableID = "inventory";
        BExpr where = new CmpExpr(EQ, "price", 0);
        Database db = jdbc.databases.get(dbId);
        Table table = db.tables.get(tableId);
        return jdbc != null
            && jdbc.databases.containsKey(dbId)
            && db.tables.containsKey(tableId)
            && table.columns.contains("price")
            && table.select(where, results);
    }
}
```

Figure 35: Declaratively initializing tests

Example2. Testing various functions of a JDBC GUI client requires a database initialized in a particular way. A tester would be interested in the behavior of a GUI under various database conditions, e.g.:

> "Does button1 work properly assuming we are connected to a database named `shop`, which has a table named `inventory`, for which when I run the query `select * from inventory where price = 0`, I get no results? What about when I get two results?"

In Fig. 35 I show a PBnJ translation of the above scenario. The `select` method is a *spec* method that selects database rows based on a given query. Declarative specifications naturally and directly capture the programmer's intent.

Underspecification – Declarative mocking is useful when the programmer does not want to implement all aspects of the functionality being mocked. Specifications naturally support this through *underspecification*. For example, if the only relevant fact about a method is that its return value is always a positive integer, this can be stated directly as a specification, and PBnJ will nondeterministically choose a value to return at run time.

```
spec boolean isWellFormedErrorInducingMsg(Msg m) {
    return !isNonErrorInducingMsg(m) && isWellFormedMsg(m);
}

spec boolean isNonErrorInducingMsg(Msg m) { ... }

spec boolean isWellFormedMsg(Msg m) { ... }
```

Figure 36: Composing specs in TFTP

Example3. To achieve the sample database initialization requirements given above without declarative mocks, the tester must manually bring the state of the mock database to the desired condition. This process involves choosing concrete values for all aspects of the database state, for example selecting exactly which two results should be returned to the given test query. In contrast, with specifications the tester does not have to specify any details about the database state other than the high-level requirements described earlier.

Nondeterminism – Similarly, nondeterministic behavior can be expressed more naturally in specifications than in manual code or a stub.

Example4. In TFTP, to test that the client side properly handles all possible scenarios of received messages, I wrote a declarative mock server and specified conditions for various types of messages it may send out to the client. The nondeterminism in the messages sent by the mock server, due to network conditions and/or server errors, was naturally captured as a logical disjunction of conditions, each specifying one legal type of message to send.

Compositionality – Logical specifications *compose* effortlessly, which allows relatively complex requirements to be expressed by composition of several simpler conditions. On the other hand, mocks implemented as imperative code do not easily compose. This was demonstrated by the TFTP application.

Example5. One class of messages that the TFTP server can send are well-formed-error-inducing (WFEI) messages. These messages could be sent due to a buggy server implementation and are used to test the error-handling behavior in the client. As seen in Fig. 36, I was able to generate messages that are WFEI simply by composing predicates for well-formed and non-error-inducing messages, whose specifications are relatively straightforward and were gleaned from the TFTP specification document [Sol92].

We cannot in general compose two different stubs in order to produce a stub that has the desired characteristics. Nor does there appear to be a natural way to employ composition in this way using imperative code. For example, both the well-formed and non-error-inducing conditions impose constraints on the block

number of a given message. Therefore, the programmer must manually deduce the range of block numbers that satisfies all constraints and then implement a method that produces block numbers in that range.

Reconfigurability of Data – Software testing often involves testing under numerous representative scenarios. One scenario may be different from another in a conceptually simple way, easily expressed by tweaking or composing logical conditions. Yet there may not be a simple way to tweak a stub representing one condition to obtain a stub representing the other. Similarly, the imperative code to produce each one may be very different.

Example6. To enable mocking of job trace and cluster data for the Hadoop application, I specified the general hierarchy of a data center cluster as well as the structure of a MapReduce job. Once the class hierarchies, integrity constraints, and specification methods were declared, I could readily produce any number of very different workload scenarios for units test with just a tweak of a few lines of specifications. When testing the Hadoop mock schedulers this proved very useful, as it enabled me to quickly produce a range of scenarios to run on.

Extensibility and Reusability – Declarative mocking is useful when the mock object is itself subject to frequent modifications or experimentation. For example, with specifications the programmer can naturally start with a bare-bones mock whose specifications are very weak and then incrementally strengthen the specifications based on the requirements of the client code under test. This kind of evolutionary process is much less natural with stubs or imperative code, since conceptually small updates to the mock's intended behavior can easily translate into tedious and sizable modifications to examples or an implementation.

Example7. I used the extensibility of specifications to my advantage in implementing the Hadoop schedulers declaratively. I first wrote the general task assignment policies (e.g. no reduce jobs can be assigned unless all map jobs are complete) in the superclass's method called `MockScheduler.assignTasksSpec()`. Then, moving to a stronger policy of FIFO, I added FIFO-specific policies for the subclass `MockScheduler_FIFO` and used the conjunction `super.assignTasksSpec() && assignTasksSpec_FIFO()` as the functional mock of the FIFO scheduler.

Disadvantages of Specifications in General: Let us now examine the flip-side of the coin.

Effort vs. Stubs – Software engineers find stub-based mock objects appealing precisely because of the very low overhead to employ them. Clearly, using logic to describe the general functionality of mocks can require substantially more developer effort.

Effort vs. Imperative Code – For small and/or simple mocks, many of the benefits of using specifications can be achieved using ordinary imperative Java code instead. For example, after writing the in-memory JDBC mock both entirely in specs and entirely in code, I realized that the SQL operations are not complex enough to justify the use of executable specifications for the purpose of mocking the functionality of a database (while they remain very useful for mocking the *data* in a database).

On the other hand, some simple algorithms like sorting on arrays and insertion into red-black trees can be succinctly specified but can be much more onerous to implement due to low-level details and subtle corner cases [SAM10]. Even our simple `MockList` example has a subtle issue when implemented imperatively: when adding an element to the list, regular Java code must check that the `elems` array has space for the new element and must allocate a bigger array if not. In contrast, the specification handles this issue implicitly.

Specifications Are Error-Prone Too – Just as in imperative code, logical specifications can be error-prone and hard to debug. Stubs are typically simple input-output pairs and so more straightforward to implement.

Efficiency and Scalability – In general constraint solving is severely limited in its efficiency and scalability versus imperative code. However, state-of-art solvers are constantly improving, and PBnJ's frame annotations help a lot in making the approach practical. In these experiments, I observed a slowdown of seconds and in a few times minutes per test. Nevertheless, during development and testing, developers may well be willing to trade off some performance for the software engineering benefits of declarative mocking illustrated here.

Limitations Specific to PBnJ: I encountered a few problems while performing these experiments that are not inherent to executable specifications, but are rather a result of limitations in the current implementation of the PBnJ tool. We reviewed the limitations specific to our choice of Kodkod as the solving backend in Chapter 4, Sec. 4.4. In performing these experiements some refactoring of code was necessary to work around these limitations.

6.4 Evaluation

Based on the exploratory study discussed in the previous section, it is clear that there are real software engineering benefits to using declarative specifications for mocking, but there are also important limitations and costs to consider. To investigate the research questions posed in Sec. 6.1, I ported the unit tests of six existing applications from Mockito to PBnJ.

6.4.1 Selection Criteria

I searched Google's open-source code repository `code.google.com` for Java applications that employ Google's Mockito library as part of their unit tests. I selected the first 6 applications in the search results whose purpose was fairly clear to me based on the descriptions, and for which I was able to gain a fair amount of understanding about the tests and the mocked components in a relatively short amount of time. I only examined unit tests that employed Mockito. I excluded tests that rely on the mock object throwing an exception, since my tool currently lacks support for specifications about exceptions. This limitation excluded 5% of the tests that use Mockito.

6.4.2 Strategy

I applied a two-phase evaluation strategy on each benchmark, which respectively address our two research questions RQ1 and RQ2 posed in the introduction:

Phase A: To learn about the overhead of using specifications and constraint solving, I first replicated each unit test by replacing existing mocks with declarative mocks that behave exactly as the developer's Mockito stubs. I wrote a separate mock class with associated PBnJ specifications mimicking the stubs for each individual test. For example, if the original stub appeared as

```
when(mockList.contains(0)).thenReturn(false)
```

then I would create a mock `List` class with the method

```
boolean contains(int x) ensures x == 0 ==> !result;
```

I compared the two versions in terms of programmer effort, expressiveness, and running time for each test.

Phase B: In the second phase, I evaluate whether access to declarative mocking could have enhanced the unit tests in terms of strength of properties tested, test coverage, and reusability, with a justifiable amount of additional effort. I consider both mocked functionality and mocked data for test initialization. For instance, in our `List` example above I would generalize the specification as follows:

```
contains(int x) ensures result <==>
some int i :  0 ..  size - 1 | elems[i] == x;
```

Unlike the first phase, here I generalized and reused a single mock class and its associated PBnJ specifications across multiple tests, as this is the natural style to use with declarative mocking.

6.4.3 Benchmarks

I examined a total of 114 unit tests among 6 benchmarks, briefly described below.

j2bugzilla is an API for interacting with a Bugzilla bug repository within Java. Unit tests mock the `BugzillaConnector` class, which uses an Apache XML RPC library to access a given Bugzilla repository.

jscep is the implementation of the Simple Certificate Enrollment Protocol (SCEP) in Java. To avoid having to test with the real objects, the unit tests mock both the certificate (an X.509 certificate in the `java.security` package), as well as the `CertificateCertifier`, the interface for verifying the identity of a given certificate.

tjays1-project1 is a personal code repository with a collection of small applications, which all employ Mockito stubs in their unit tests.

gcm-server is the server side implementation of cloud messaging service for Android. Google developers use a mock of the `Sender` class, responsible for sending messages over an HTTP connection, to test any number of possible scenarios.

shivaminesweeper is a servlet-based Minesweeper game running in the browser. Unit tests mock the HTTP connection and stub the requested parameters to verify the functionality of the game implementation based on user events.

birthdefectstracker is a web-based application to query and manipulate a database of medical records. The most notable use of mock objects is that of various Data Access Objects (DAOs). Unit tests are generally used to test proper behavior of DAO controllers, for example that an error is signaled when a username is entered that already exists in the database.

6.4.4 Phase A Results

In phase *A* all unit tests were successfully refactored to replace stubs with specs that retain their exact behavior. Table 9 summarizes the results for both phases of the evaluation[15]. As a rough measure of developer "effort" required to replace stubs with specifications, I compared the number of lines between the stub and spec versions. The third column of Table 9 shows that on average there was a 2:1 lines-of-code ratio between specs and stubs, respectively. The average slowdown due to constraint solving was one second per test. Based on this data, there is considerable effort overhead for the tester to employ specifications

[15]Solving on a Core i7-3930K, 3.20GHz, with 8-bit integers.

merely as stubs, which is to be expected since input-output stubs can make very little use of the benefits of specifications.

The functionality of Mockito's `verify()` (which tracks invocation counts for a stubbed method) cannot be directly replicated using specifications. I replicated this task indirectly by declaring auxiliary counter fields and adding additional assertions in the postconditions to increment these counters on each invocation. In many cases there was a more direct property to be checked which would obviate the need for simulating `verify()`; phase *B* below explores that possibility.

6.4.5 Phase B Results

In phase *B* I revisited each test to examine whether the specifications from the previous phase could be generalized to take advantage of the benefits of declarative mocking illustrated in Sec. 6.3. Below I sample some of the positive and negative scenarios that I encountered.

When Specifications Were Useful:

(D) Data Mocking, Data Integrity – In *shivaminesweeper* there are many implicit relationships among objects representing various aspects of the game, such as the dimensions of the board, the mine count for each cell on the board, etc. Many tests set up the game board manually by constructing a specific game state. I instead specified the relationships between various objects using object invariants. This simplified the task of test initialization. For example, once the dimensions for the board are provided, the invariants automatically determine the appropriate number of mines to include and place them on the board nondeterministically. This use of invariants prevents the creation of inconsistent states, which are easy to accidentally introduce when initializing state manually.

(R) Reusability – In *birthdefectstracker* I removed the existing stubs and reused the database specifications from the JDBC mock from the exploratory study to generalize each test. I used data mocking to initialize various snapshots of each database declaratively, and recycled the initialization conditions from one test to another to reduce effort.

(U) Underspecification – One of the applications in *tjays1-project1* is an implementation of an elevator unit, where tests verify that the implementation chooses the right floor to stop at next. An object keeping a priority set of floors that have requested service is mocked. Instead of hardcoding a set of floors as done in the original stubs, I request an underspecified set of floors, and employ the `unique` modifier to test a variety

```
Sender sender = Mock(Sender.class);
Result message = new Result();
doReturn(null) // fails 1st time
  .doReturn(null) // fails 2nd time
     .doReturn(result) // succeeds 3rd time
        .when(sender).send(message, "1");
```

Figure 37: Use of stubs in *gcm* for simulating a scenario that includes failures and success

```
class MockSenser extends Sender {
  spec int sendCount;
  unique Result send(Message msg, String id)
    ensures sendCount == this.old.sendCount + 1
      // up to 4 times ok/fail nondeterministic:
      && (result != null || this.old.sendCount < 5);
}
```

Figure 38: Specifications generalize Fig. 37 scenario.

of context within a single test, with only minimal modifications to the original unit tests. Here, specifications increase the coverage of each test while requiring the same amount of developer effort.

(N) Nondeterminism – Several tests in *gcm-server* check that the `send` method properly retries message sends whenever they get dropped. Fig. 37 shows how Google developers use Mockito's cascaded stub feature to test particular scenarios involving dropped messages. Specifications express this nondeterminism naturally, as a disjunction of possible outcomes, as illustrated in Fig. 38. I added the `unique` modifier to generalize tests such as this to cover any number of possible outcomes, making several other existing tests redundant. Writing these specs does not require much more effort than the stubs in Fig. 37.

When Specifications Were Not Useful:

(1) When Stub Is Irrelevant to the Test – In the *jscep* benchmark, as shown by the example in Fig. 39, unit tests use stubs to check that the certificate certifier is properly invoked by the client code under various circumstances. This represents a case where specifications do not enhance this test in any way, as there is no relation between the property being tested (the certifier has been properly invoked) and the logic of the stubbed components (the certifier's behavior).

(2) When Mocking Functionality Is Simple with Code – As we discovered in the exploratory study (during mocking of SQL operations) sometimes mocking an object's behavior is straightforward using imperative code and the overheads of using specifications and run-time constraint solving are not justifiable.

Results: The last 6 columns in Table 9 report the result of the second phase of the evaluation. I state the percentage of examined unit tests for each benchmark that were enhanced by declarative data mocking and

```
void testHandlerForCertificate() {
    certifier = mock(Certifier.class);
    cert = mock(X509Certificate.class);
    when(certifier.certify(cert)).thenReturn(true);
    // perform handler test here...
    // verify certifier's certify method was invoked:
    verify(certifier).certify(cert);
}
```

Figure 39: *jscep* example where specs not useful

Table 9: Declarative mocking benchmark data

		Phase A		Phase B					
application	**#tests with stubs**	**spec/stub LoC ratio**	**avg/worst time (sec.)**	**% (D)**	**% (R)**	**% (NU)**	**%tests enhanced**	**spec/stub LoC ratio**	**avg/worst time (sec.)**
j2bugzilla	13	1.4	4/10	77	85	69	85	0.4	12/95
jscep	4	2.6	0/0	0	0	0	0	–	–
tjays1-project1	18	1.8	1/2	44	44	39	44	0.8	1/2
gcm-server	23	0.9	1/2	22	30	30	30	1.0	2/3
shivaminesweeper	15	1.8	1/2	93	93	93	93	0.7	52/64
birthdefectstracker	41	2.8	0/1	73	73	66	73	1.9	104/335

data integrity (D), reuse and reconfiguration (R), and nondeterminism and underspecification (NU). Clearly, many tests belong to multiple categories, and some of properties I mentioned in Sec. 6.3 were left out due to being difficult to accurately quantify. I dub tests that exhibit at least one of these properties as "enhanced." The next column indicates that 54% of all unit tests were able to be enhanced in this way.

In performing this experiment and studying the results, I observed a general pattern. Among unit tests where there was a strong relationship between the logic of the unit test and the stubbed component, declarative mocking of functionality was typically beneficial. On the other hand, when mocks were simply there to enable running of the tests, with no direct relation to the properties being tested, stubs were sufficient and specifications were not worth the effort. Declarative data mocking, on the other hand, was typically beneficial any time there existed complex test initialization data.

The second-to-last column compares the lines of code as a rough measure of developer "effort," among those tests that benefited from declarative mocking. Because the specifications were generally reusable across the tests for a given application, this ratio dropped to 1:1 on average. Thus employing specifications can produce their many benefits for the purpose of mocking, while requiring comparable amount of developer efforts over a test suite when compared to traditional approaches.

The last column reports on constraint solving times by Kodkod. The average solving time in Phase *B* was 34 seconds per test. As I mentioned, this solver works by direct translation to SAT and constraints involving a lot of integer arithmetic can take a long time to solve. This was the case in both *shivaminesweeper* and

birthdefectstracker, where specifications involved arithmetic constraints over the elements of a multidimensional array. The current PBnJ tool is not optimized for these situations.

6.5 Related Work

I now briefly overview the literature.

6.5.1 Mock Objects

Several libraries are designed to allow testers to produce simple mock objects, including Mockito [Fab], Mockrunner [AI], and Microsoft Moles [dHT10]. These frameworks make traditional stub-based mock objects easier to create, while this work focuses on making mock objects more expressive and declarative. Ostermann incorporates nondeterministic choice to make mock objects more expressive [AO10]; declarative mocking naturally supports nondeterminism as well as additional expressiveness.

Saff *et al.* [SAPE05] propose an approach to automatically create mock objects for the purpose of test factoring by capturing the interactions between a component and its environment on a set of system-wide tests. This approach requires that the full system be available initially. Similarly, Qi *et al.*'s method [QSQ+12] creates environment models based on execution traces, so it also requires a fully executable version of the program including of the mocked environment. My approach does not have this limitation and allows more control over what properties of the environment to mock, but it requires explicit specifications.

In prorogued programming [ABS12], the system interactively asks the user to supply an appropriate return value upon a call to the method. The supplied values are recorded for later use, which has the effect of incrementally building up an appropriate mock for the method.

Henkel *et al.*'s approach [HRD08] is conceptually similar to this work but uses term rewriting on specifications rather than constraint solving for producing mocks. Their approach relies on heuristics to guide the rewriting, which can miss solutions and/or lead to infinite search, while our specifications are more general, and soundness and completeness are guaranteed, up to the search bounds. Wilmore [WE06] proposes an automatic database state preparation approach for test initialization of database applications via intensional specifications as constrained queries. This work can be thought of as an instance of declarative mocking of data, and my approach can handle it, as evidenced by the JDBC and MapReduce examples.

As mentioned in Sec. 6.1, declarative mocking uses similar technology to prior work on automated test generation, but with distinct goals. Closest to my work is prior research that automatically produces mock objects

for use with generated tests. For instance, Galler *et al.* [GMW10] generate test inputs by automatically extracting mock object stubs that satisfy user-specified preconditions. However, these mock objects are limited in expressiveness; for example, the values returned from mocked methods are determined statically and may not depend on the inputs or state of the object under test. My approach solves constraints dynamically and so does not suffer from these limitations.

6.5.2 Declarative Execution

The idea of employing a *mixed interpreter* for mock objects was mentioned by Rayside *et al.* [RMY+09b], yet the idea was not investigated concretely.

I use PBnJ [SAM10] to enable declarative mocking. Other recent declarative execution systems including Squander [MRYJ11] and Kaplan [KKS12] (reviewed in Sec. 4.3.1) may equally be used.

6.6 Conclusions

I have presented a new approach to creating mock objects. Programmers write high-level specifications for the methods in an API being mocked, and a constraint solver dynamically executes these specifications. As a result, code that depends on the API can be tested exactly as if it is invoking a "real" implementation of the API. Further, I show that executable specifications naturally support other testing tasks, in particular the initialization of state for both the object under test as well as the mocked objects.

I extended PBnJ to better support declarative mocking, and have used the implementation both to explore the potential capabilities of the approach as well as to directly compare with the usage of traditional mock objects on existing applications. Declarative mocking of behavior is most beneficial for unit tests where there is a strong relation between the logic of the unit test and the stubbed component, and declarative data mocking can often simplify initialization code and increase test coverage.

This study suggests that the traditional and declarative mocking approaches are complementary and both are useful in particular scenarios during testing. Therefore I think mock libraries should provide both facilities of traditional and declarative mocks, and programmers will find scenarios where this approach offers practical benefits for them.

Chapter 7

7 Conclusion

In the field of programming languages and systems, the study of *formal verification methods* is pervasive. There are very many researchers who dedicate their entire work to verification and the question of "*is there a bug in the system.*" The verification community has devised and deployed powerful mathematical and logical decision procedures for this purpose.

What I have tried to argue in my research is the following.

1. Verification is not the end of it. In fact, it may sometimes be the simpler task compared to what should come next, which is "*how to fix it.*" Why is this often very challenging, time-consuming, and error-prone question left to the poor souls of our developers to figure out by themselves? I am aware that many researchers have been and are working on the problems of automated program synthesis and repair, but from what I have seen the scale is heavily skewed towards the first question.

2. The same powerful mathematical and logical decision procedures and constraint solving techniques can in fact aid us towards the second question. We can overcome their inefficiencies by zooming in and focusing on particular scenarios and domains. There probably isn't going to be a "one-size-fits-all" solution. Let's not pretend.

In this dissertation, I argued for leveraging the efforts that our hard working software engineers continue to put into validating their code, along with the powerful formal techniques our good researchers have come up with for this purpose, towards novel software engineering benefits. To this point, I demonstrated two new practical applications of declarative execution in software engineering: online failure recovery and declarative mocking, as well as one of the first cases of automated code repair with completeness guarantees.

Based on these works, I now more believe in the motto "*do less, but do it right.*" The same "impractical" techniques of static and run-time constraint solving, declarative programming, etc. will find more practical applications as we learn how to narrow our focus and try new angles, domains, and contexts.

References

[ABS12] Mehrdad Afshari, Earl T. Barr, and Zhendong Su. Liberating the programmer with prorogued programming. In *Onward! '12*, pages 11–26, New York, NY, USA, 2012. ACM.

[AI] Gabor Liptak Alwin Ibba, Jeremy Whitlock. Mockrunner. http://mockrunner.sourceforge.net.

[AKD+10] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE TSE*, 36(4):474–494, 2010.

[AO10] Michael Achenbach and Klaus Ostermann. Testing object-oriented programs using dynamic aspects and non-determinism. In *ETOOS '10*, pages 3:1–3:6, New York, NY, USA, 2010. ACM.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[Blo] Personal Blog. Headbirths. http://headbirths.wordpress.com/2012/02/16/alief-belief-and-c-lief.

[BLS05a] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

[BLS05b] Mike Barnett, Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS '05*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

[CGP08] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. Self-healing by means of automatic workarounds. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 17–24, New York, NY, USA, 2008. ACM.

[CS98] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, pages 48–59, New York, NY, USA, 1998. ACM.

[CTBB11] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. Angelic Debugging. In *ICSE*, pages 121–130, 2011.

[DCJ06] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with sat. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 109–120, New York, NY, USA, 2006. ACM.

[dHT10] Jonathan de Halleux and Nikolai Tillmann. Moles: tool-assisted environment isolation with closures. In *Proceedings of the 48th international conference on Objects, models, components, patterns*, TOOLS'10, pages 253–270, Berlin, Heidelberg, 2010. Springer-Verlag.

[DKTE04] Alan Donovan, Adam Kieżun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java Programs to Use Generic Libraries. In *OOPSLA*, pages 15–34, 2004.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DR03] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 78–95, New York, NY, USA, 2003. ACM.

[DR05] Brian Demsky and Martin C. Rinard. Data structure repair using goal-directed reasoning. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 176–185. ACM, 2005.

[DYJ08] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 130–145, Berlin, Heidelberg, 2008. Springer-Verlag.

[EGSK07] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer 0002, editors, *ASE*, pages 64–73. ACM, 2007.

[EK08] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: a tool for repairing complex data structures. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 855–858, New York, NY, USA, 2008. ACM.

[EKVM07] Bassem Elkarablieh, Sarfraz Khurshid, Duy Vu, and Kathryn S. McKinley. Starc: static analysis for efficient repair of complex data. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 387–404, New York, NY, USA, 2007. ACM.

[Fab] Szczepan Faber. Mockito: Simpler and better mocking. http://code.google.com/p/mockito.

[FBB92] Bjørn N. Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In Ole Lehrmann Madsen, editor, *ECOOP*, volume 615 of *Lecture Notes in Computer Science*, pages 268–286. Springer, 1992.

[FF01] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–15, New York, NY, USA, 2001. ACM.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[FMPW04] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, objects. In *OOPSLA '04*, pages 236–246, New York, NY, USA, 2004. ACM.

[Fow] Martin Fowler. Mocks Aren't Stubs. http://martinfowler.com/articles/mocksArentStubs.html.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[GKA+11] Vijay Ganesh, Adam Kieżun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection. In *CAV*, pages 1–19, 2011.

[GMW10] Stefan J. Galler, Andreas Maller, and Franz Wotawa. Automatically extracting mock object behavior from design by contract specification for test data generation. In *AST '10*, pages 43–50, New York, NY, USA, 2010. ACM.

[Gul11] Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples. In *POPL*, pages 317–330, 2011.

[had] Apache Hadoop. http://hadoop.apache.org.

[HRD08] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Developing and debugging algebraic specifications for java classes. *ACM Trans. Softw. Eng. Methodol.*, 17(3):14:1–14:37, June 2008.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[JCh] JChessBoard. http://jchessboard.sourceforge.net.

[JDB] JDBC. Oracle Corporation. http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc.

[Jon] M. Tim Jones. Scheduling in hadoop. http://www.ibm.com/developerworks/linux/library/os-hadoop-scheduling/index.html.

[jst] JStock. http://jstock.sourceforge.net.

[JV00] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 14–25, New York, NY, USA, 2000. ACM.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[KKS12] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *POPL '12*, pages 151–164, New York, NY, USA, 2012. ACM.

[KM04] Sarfraz Khurshid and Darko Marinov. Testera: Specification-based testing of java programs using sat. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[KMJ02] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 231–245, New York, NY, USA, 2002. ACM.

[KTH09] Soonho Kong, Nikolai Tillmann, and Jonathan de Halleux. Automated testing of environment-dependent programs - a case study of modeling the file system for pex. In *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, ITNG '09, pages 758–762, Washington, DC, USA, 2009. IEEE Computer Society.

[KW06] Ben Krause and Tim Wahls. jmle: A tool for executing jml specifications via constraint programming. In *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 293–296. Springer, 2006.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[Lei07] Rustan M. Leino. Specifying and verifying software. In *ASE '07*, pages 2–2, New York, NY, USA, 2007. ACM.

[Mey97a] Bertrand Meyer. Design by contract: Making object-oriented programs that work. In *TOOLS (25)*, page 360, 1997.

[Mey97b] Bertrand Meyer. Design by contract: Making object-oriented programs that work. In *TOOLS (25)*, page 360. IEEE Computer Society, 1997.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. Extreme programming examined. chapter Endo-testing: unit testing with mock objects, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Min86] Marvin Minsky. *The society of mind*. Simon & Schuster, Inc., New York, NY, USA, 1986.

[Min05] Yasuhiko Minamide. Static Approximation of Dynamically Generated Web Pages. In *WWW*, pages 432–441, 2005.

[MKM13] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *ICSE*, 2013.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[Mor88] Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[MRYJ11] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code. In *ICSE '11*, pages 511–520, New York, NY, USA, 2011. ACM.

[MS11] Anders Møller and Mathias Schwarz. HTML Validation of Context-Free Languages. In *FOSSACS*, pages 426–440, 2011.

[MT06] Yasuhiko Minamide and Akihiko Tozawa. XML Validation for Context-Free Grammars. In *APLAS*, pages 357–373, 2006.

[MXT+09] Madhuri R. Marri, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. An empirical study of testing file-system-dependent software with mock objects. In *AST'09*, pages 149–153, 2009.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[NNNN11] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Auto-Locating and Fix-Propagating for HTML Validation Errors to PHP Server-Side Code. In *ASE*, pages 13–22, 2011.

[NQRC13] Hoang D. T. Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis. In *ICSE*, 2013.

[NZGKM12] Razieh Nokhbeh Zaeem, Divya Gopinath, Sarfraz Khurshid, and Kathryn S. McKinley. History-aware data structure repair using sat. In *Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 2–17, Berlin, Heidelberg, 2012. Springer-Verlag.

[QSQ+12] Dawei Qi, W.N. Sumner, Feng Qin, Mai Zheng, Xiangyu Zhang, and A. Roychoudhury. Modeling software execution environment. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 415–424, 2012.

[Rin12] Martin Rinard. What to do when things go wrong: recovery in complex (computer) systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development Companion*, AOSD Companion '12, pages 1–2, New York, NY, USA, 2012. ACM.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[RMY+09a] Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson. Agile specifications. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 999–1006, New York, NY, USA, 2009. ACM.

[RMY+09b] Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson. Agile specifications. In *OOPSLA '09*, pages 999–1006, New York, NY, USA, 2009. ACM.

[SAH+10] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *IEEE Symp. on Security and Privacy*, pages 513–528, 2010.

[SAM10] Hesam Samimi, Ei Darli Aung, and Todd Millstein. Falling back on executable specifications. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 552–576. Springer-Verlag, Berlin, Heidelberg, 2010.

[SAPE05] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for java. In *ASE '05*, pages 114–123, New York, NY, USA, 2005. ACM.

[SLJB08] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching Concurrent Data Structures. In *PLDI*, pages 136–148, 2008.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *ASPLOS*, pages 404–415, 2006.

[Sol92] Karen R. Sollins. The TFTP protocol (revision 2). *Internet RFC 1350*, July 1992.

[SSA+12] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[ST09] Friedrich Steimann and Andreas Thies. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In *ECOOP*, pages 419–443, 2009.

[STL] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Gecode: generic constraint development environment. http://www.gecode.org.

[Swa] Brian Swan. Mocks suck (and what to do about it). https://www.engineyard.com/video/16285089.

[Swe] SweetHome3D. http://www.sweethome3d.eu.

[Tan] Hong Tang. Mumak: Map-reduce simulator. https://issues.apache.org/jira/browse/MAPREDUCE-728.

[TFK+11] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring Using Type Constraints. *ACM TOPLAS*, 33(3), 2011.

[Tor09] Emina Torlak. *A constraint solver for software engineering: Finding models and cores of large relational specifications*. Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[TS06] Nikolai Tillmann and Wolfram Schulte. Mock-object generation with behavior. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 365–368, Washington, DC, USA, 2006. IEEE Computer Society.

[VJ03] Mandana Vaziri and Daniel Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 505–520. Springer, 2003.

[W3T] W3Techs. Usage Statistics and Market Share of PHP for Websites. http://w3techs.com.

[WBMG11] Guanying Wang, Ali R. Butt, Henry Monti, and Karan Gupta. Towards synthesizing realistic workload traces for studying the hadoop ecosystem. In *MASCOTS '11*, pages 400–408, Washington, DC, USA, 2011. IEEE Computer Society.

[WE06] David Willmor and Suzanne M. Embury. An intensional approach to the specification of test cases for database applications. In *ICSE '06*, pages 102–111, New York, NY, USA, 2006. ACM.

[WGSD07] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. *ACM TOSEM*, 16, September 2007.

[WLB00] Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing formal specifications with concurrent constraint programming. *Automated Software Engg.*, 7(4):315–343, 2000.

[WNGF09] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In *ICSE*, pages 364–374, 2009.

[WS08] Gary Wassermann and Zhendong Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *ICSE*, pages 171–180, 2008.

[YAB11] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Patching Vulnerabilities with Sanitization Synthesis. In *ICSE*, pages 251–260, 2011.

[ZKR08] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 349–361. ACM, 2008.

[ZKR09] Karen Zee, Viktor Kuncak, and Martin C. Rinard. An integrated proof language for imperative programs. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–351. ACM, 2009.

[ZML+12] Linghao Zhang, Xiaoxing Ma, Jian Lu, Tao Xie, N. Tillmann, and P. de Halleux. Environmental modeling for automated cloud application testing. *Software, IEEE*, 29(2):30 –35, march-april 2012.