The Dissertation Committee for Divya Gopinath
certifies that this is the approved version of the following dissertation:

Systematic Techniques for More Effective Fault Localization and Program Repair

Committee:

Sarfraz Khurshid, Supervisor

Dewayne Perry

Keshav Pingali

Christine Julien

Randolph Bias

Systematic Techniques for More Effective Fault Localization and Program Repair

by

Divya Gopinath, B.Tech., M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2015

Acknowledgments

I consider myself very fortunate to have been able to pursue my research under the guidance of Dr. Sarfraz Khurshid. Not only has he been a constant source of encouragement and motivation but is one of the very few people I have met who are so compassionate and understanding of other's situations. I am grateful for all the help, advice and guidance that he has provided me and hope to be in constant collaboration with him in the future.

I am privileged to have an esteemed dissertation panel and would like to thank Dr. Perry, Dr. Pingali, Dr. Julien and Dr. Bias for their timely feedback and inputs that has lent a new perspective to my research. I hope to learn more from your association in the future as well. All my colleagues from the SVVAT team, have been a source of inspiration. I would like to specifically thank Razieh Nokhbeh, Zubair Malik, Ripon Saha, Nima Dini, Kaiyuan Wang, Lingming Zhang, Shadi Abdul Khalek, Danhua Shao, and Gouwei Yang, for their collaboration with me and the help they have provided me on many occasions.

My biggest asset has been my family; my husband, parents and parents-in-law, without whose support it would have been impossible for me to proceed on the path to a PhD, bearing a child along the way! Lastly, but most importantly, I would like to thank my lucky charm who literally *kick* started my publishing streak. I am blessed to have my little boy, Nikhil, who has been ever so adjusting of changing

situations and bearing with his mom's absence at many occasions.

The projects included in my research were funded by the following grants, NSF Grant Nos. IIS-0438967 and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

Systematic Techniques for More Effective Fault Localization and Program Repair

Publication No. _____

Divya Gopinath, Ph.D.
The University of Texas at Austin, 2015

Supervisor: Sarfraz Khurshid

Debugging faulty code is a tedious process that is often quite expensive and can require much manual effort. Developers typically perform debugging in two key steps: (1) *fault localization*, i.e., identifying the location of faulty line(s) of code; and (2) *program repair*, i.e., modifying the code to remove the fault(s). Automating debugging to reduce its cost has been the focus of a number of research projects during the last decade, which have introduced a variety of techniques. However, existing techniques suffer from two basic limitations. One, they lack accuracy to handle real programs. Two, they focus on automating only one of the two key steps, thereby leaving the other key step to the developer.

Our thesis is that an approach that integrates systematic search based on state-of-the-art constraint solvers with techniques to analyze artifacts that describe application specific properties and behaviors, provides the basis for developing more effective debugging techniques. We focus on faults in programs that operate

on structurally complex inputs, such as heap-allocated data or relational databases. Our approach lays the foundation for a unified framework for localization and repair of faults in programs. We embody our thesis in a suite of integrated techniques based on propositional satisfiability solving, correctness specifications analysis, test spectra analysis, and rule-learning algorithms from machine learning, implement them as a prototype tool-set, and evaluate them using several subject programs.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems continue to be plagued by failures due to defects introduced during software development. According to a NIST study from 2002 [84], software errors cost the U.S economy an estimated 59.5 billion dollars annually. Around 50% to 80% of software development and maintenance effort is spent in detecting and fixing bugs in programs [28]. The main reason for this process being so expensive and time consuming is that it is largely manual, specifically the exercise of locating faulty portions of code and correcting them. More effective techniques to automate these activities are badly needed.

Figure 1.1 presents an overview of the process of detecting and correcting faults in programs. The basic inputs that need to be provided by the user are the program to be tested and a correctness criteria that describes the expected behavior of the program. The correctness criteria could be in the form of *test cases*, consisting of a pre-defined set of concrete inputs and the corresponding expected outputs, or *specifications* that describe the correctness properties that need to be satisfied by the program.

The process of *testing* aims to find failures of expected properties. Considerable amount of work has been done to automate testing [12, 23, 24, 39, 41, 42, 49,

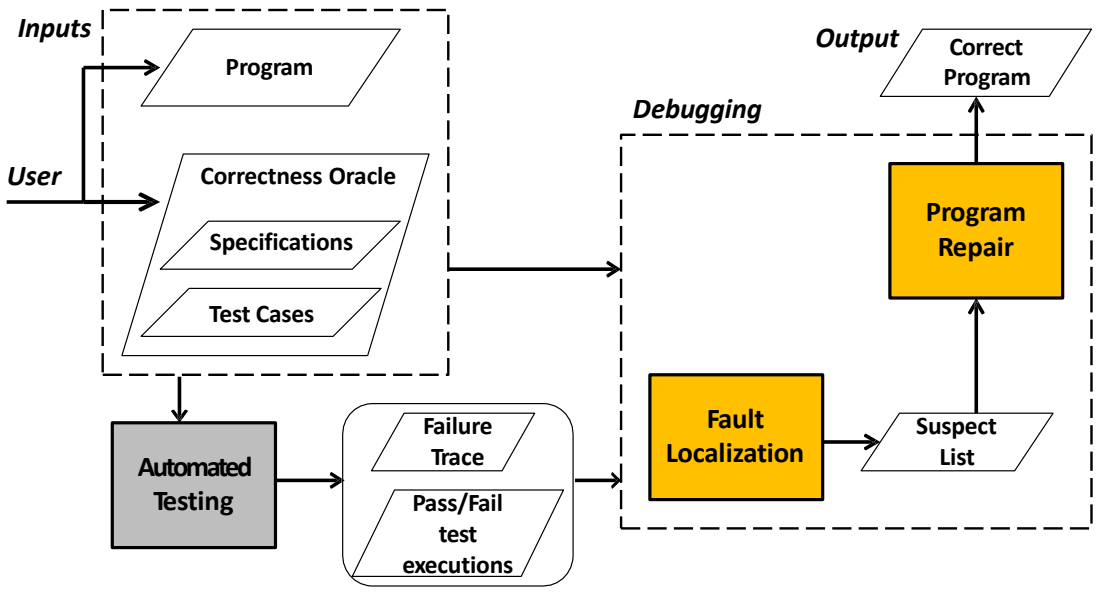

Figure 1.1: Testing and Debugging framework

55, 56, 71, 72, 94, 105, 112], The process of *debugging* aims to remove the fault(s) in the program that cause erroneous executions. There are two main steps involved in debugging: i) identifying the location of faults in code, which is termed as *fault localization*, and ii) modifying the code to remove the fault(s), which is termed as *program repair*. Both these activities are traditionally manual and expensive. Our focus is on providing accurate and efficient solutions to automate them.

1.1 Problem Description

Fault localization is the problem of determining a subset of the statements of the program such that the program can be corrected by modifying some or all of the selected statements [27, 64, 90, 110]. Such a subset is called a *suspect list*, which is usually ranked based on the likelihood of each statement in the set being

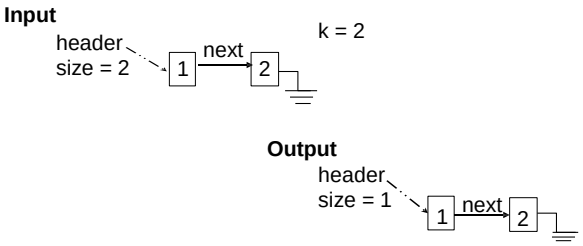

Figure 1.2: Failing Test.

faulty. *Program repair* is the problem of modifying the given program p, into another program p' such that p' is correct with respect to the given correctness criteria. A suspect list can be used to focus repair on specific parts of the program.

Our focus is on faults that lead to the violation of behavioral correctness properties. Debugging such faults requires careful analysis of the code, state and the correctness specifications. This necessitates considerable user-involvement which makes automating the process of debugging more challenging. We specifically consider programs dealing with data structures and relational tables, which form the core of many parsers, compilers, model-finders, and data-processing applications today. Data structures such as linked lists, binary search trees etc., have complex state configurations allocated on the heap and exhibit rich properties of structural integrity such as acyclicity. Debugging violations of complex behaviors and properties becomes very time-consuming when done manually. It is equally challenging to debug programs which deal with large amount of tabular data and display erroneous behavior only on specific records in the tables. Most previous techniques to automate fault localization and repair have not been applied on such programs.

1.1.1 Example

```
pred repOk(This: SLL){ //class invariant of SLL
   all n: This.header.*next | n !in n.^next //acyclicity
   # This.header.*next = int This.size //size-invariant
   all n, m: This.header.*next |
      int m.key = int n.key => n = m } //unique-elements

pred deletepostcond(This: SLL, k: Int){
  //remove-ok
  This.header.*next.key - k = This.header`.*next`.key`
  && ((k in (This.header.*next.key))
           => (This.size - 1 = This.size`)) } //size-ok
```

Figure 1.3: Correctness constraints in Alloy.

```
class LinkedList{
      Node header;
      int size; // number of nodes in this list
      static class Node{
         Node next;
         int key;}
      boolean delete(int k){
         // removes the node with key value k
 0:      Node prev = null;
 1:      Node lst = this.header;
 2:      while(lst != null){
 3:         if(lst.key == k){
 4:            if(prev != null){
 5:               prev.next = lst.next; //correct
                    prev.next = lst; //removeErr
               }else{
 6:               this.header = this.header.next;
               }
 7:            this.size--;
 8:            return true;}
 9:         prev = lst;
10:         lst = lst.next;}
11:      return false;}}
```

Listing 1.1: Buggy delete method of Singly Linked List.

Listing 1.1 is a representative example of the type of faults we deal with. A singly linked list data structure (SLL) comprises of nodes with integer elements, linked to each other by `next` pointers. The data structure is characterized by its structural properties: there should not be any cycles or loops when traversing the list using the `next` pointers, each node should have a unique integer value, and the `size` field should contain the correct count of nodes in the list. The `delete` method

attempts to remove from the list, the node whose value matches the input. The fault in statement 5 leads to an output where the node with the matching element is not removed from the list.

Recent advances in specification languages [19, 54] and program annotations aid users in writing rich specifications of correctness. Figure 1.3 presents the correctness specifications for the SLL `delete` method written in the Alloy language [58] (Chapter 2). The correctness oracle for the method expressed as the post-condition of the procedure, ensures that the rich structural invariants of the data structure (`repOk` in Figure 1.3) are satisfied in addition to checking the behavior of the `delete` in removing the correct node from the list (`deletepostcond` in Figure 1.3). Figure 1.2 shows a test input that would expose the fault in the code and the corresponding incorrect output. The `remove-ok` constraint, which checks that the input value `k` should not be present in the output list, is an example of the correctness property that this output structure violates. Our goal is to debug this program by identifying the faulty statement 5 (`prev.next = lst`) and replacing it with the correct statement (`prev.next = lst.next`).

1.1.2 Challenges in localizing faults

Traditionally developers would manually place print statements or use debugging aids to place breakpoints to determine faulty statements by observing values of state variables. Such analysis is very time consuming. Program Slicing [110] is one of the oldest approaches that attempts to statically determine portions of code that could impact the erroneous state variables. However, the technique tends to conservatively produce long suspect lists. .

Fault localization approaches that are fast, practical and hence more popularly used are those which take advantage of multiple test runs of a program [60, 73]. Spectra-based localization [64] is a popular approach that uses dynamic test case execution information such as statement coverage of passing and failing runs to rank code entities in order of suspiciousness or likelihood of being faulty. However, the number and coverage of the tests can adversely impact the localization. For instance, many failing tests with similar code coverage or passing tests having coverage very different from the failing tests can result in too many statements being marked suspicious of being faulty. Analyzing manually even 5% of a 100K LOC (which is the effectiveness of the most popular fault localization approach [64]) can be a daunting task.

The type and number of faults also affects the accuracy of localization based on test coverage. Certain faulty statements may get covered by both failing and passing tests resulting in them being ranked less suspicious than an actually correct statement being covered by failing tests alone. For instance, on statement 7 in our example, erroneously setting `size` to 0 instead of decrementing it, would result in a wrong output only for lists having more than one nodes. Failing tests covering different faults in the same code, may interfere with each other leading to a decrease in the suspiciousness values assigned to the faulty statements.

Another issue with existing localization approaches is that parsing through arbitrary statements with high suspiciousness value does not provide developers with the context that would help them investigate the cause for the failure. In practise, developers mostly revert back to the traditional method of manual debugging

after parsing through the top few statements in the suspect list [95]. Hence there is a pressing need to improve the effectiveness of localization approaches.

1.1.3 Challenges in correcting faults

Even if the exact location of the fault is known, correcting it is still challenging. A naive method is to apply simple modifications to the statement. However, in the presence of complicated operations and correctness properties, the user needs a lot of expertise to explore the options for altering the statement locally and analyze its global impact. Another approach to the problem is exhaustive generation of all combinatorial possibilities of syntactically valid variants of the original statement. Each variant would then need to be validated against the correctness oracle. Genetic programming [108] is a recently proposed approach that uses a population based search heuristic to guide the search for a variant that passes all the tests of the program. However, such techniques tend to become intractable and inaccurate as the number of possible variants increases.

Automated program synthesis techniques address a similar problem [69, 100]. However, the tradition of synthesis is to generate a program from scratch or complete an implementation sketch but not to alter an existing faulty implementation. As part of repair, we do not intend to replace the existing implementation with a completely different one, but to make modifications to the faulty statements that would correct the behavior.

An effective method to repair programs, specifically those that manipulate structurally complex data that pervade modern software, remains a challenge.

1.1.4 Characteristics of an Effective Solution

There are two main parameters that determine the effectiveness of an approach for debugging,

- **Accuracy:**

  A fault localization approach is accurate if:

  1. It produces a suspect list that does not miss any faulty statement.
  2. It produces a precise suspect list that contains only the faulty statements.
  3. It ranks the faulty statements highest in the list.

  A program repair approach is accurate if:

  1. It yields the correct output for the failing executions and does not introduce new faults.
  2. It is able to reason about complex correctness properties correctly.
  3. It produces alterations that are generalizable and work correctly on future inputs as well.

- **Efficiency:**

  An automated debugging approach is efficient if it can locate and repair faults in programs with complex behaviors, properties and large state-spaces within a reasonable amount of time.

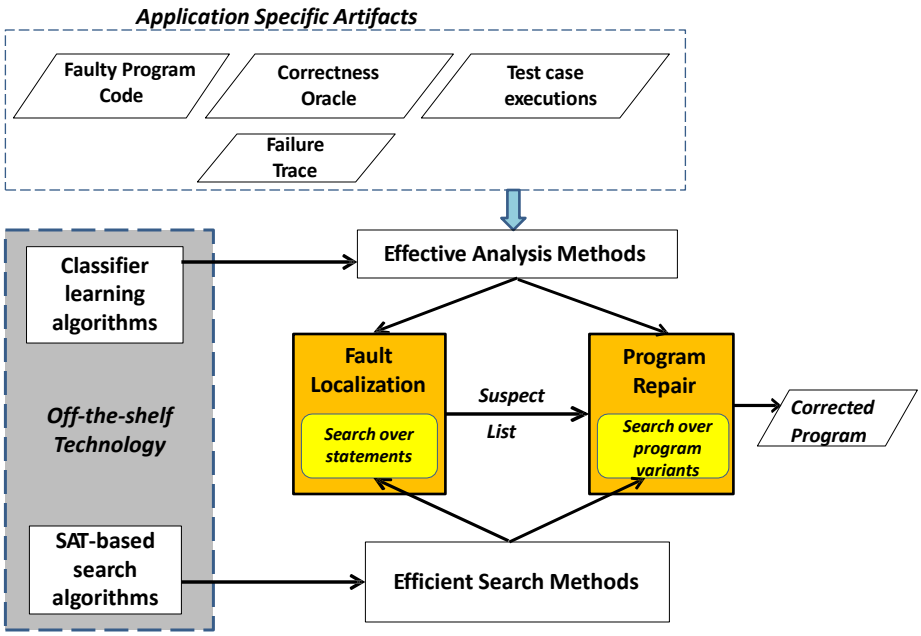

Figure 1.4: Our solution framework

1.2 Our Solution

Our solution focuses on two research thrusts;

- *Methods to search* through the space of program statements and their possible variants.
- *Methods to analyze* application-specific artifacts such as correctness specifications and test case executions to infer properties that would help guide the search and improve the effectiveness of debugging.

We provide a novel combination of ideas in both these thrusts to develop more effective approaches for fault localization and program repair.

Figure 1.4 presents a diagrammatic representation of our solution. The core functionality of both fault localization and program repair is *search* — search over the statements of the program to determine the faulty ones, and search over possible program variants to determine a correct version. We leverage off-the-shelf boolean satisfiability solvers to perform systematic search over the space of possibilities. Complex programs with multiple paths and a large state-space can lead to a huge search space but recent advances in SAT technology (Section 4.3.1) can often handle the resulting combinatorics. The problem of localizing faulty portions of the code and looking for a variant that satisfies the correctness oracle is cast as a formula in boolean logic. The control-flow, data-flow and the state of a procedure are encoded as constraints in relational logic and then translated down to boolean satisfiability [34, 104]. Representing imperative code in stateless declarative logic enables goal-directed reasoning of multiple program paths.

Artifacts such as the program source code, test cases, and correctness specifications are rich sources of application-specific knowledge that can aid in improving the effectiveness of generic search algorithms for the particular problem under consideration. For instance, if the suite used for testing has a large number of failing and passing tests with sufficient diversity in their code and state coverage, statistical analysis of their profiles could point to the actually faulty statements with high confidence. This information could greatly aid in speeding up the search process for localization. Similarly, rich user-defined specifications of correctness on the state of a program can guide the search for the program variant that would produce the correct output. Hence, we employ effective analysis methods to deduce application-

specific behaviors and properties from available resources and use it to customize the search for the specific problem at hand. We have explored the application of typical algorithms from machine learning [81, 88] for effective analysis.

1.2.1 Our thesis

Our thesis is that an approach that integrates systematic search based on state-of-the-art constraint solvers [36, 102] with techniques to analyze artifacts (such as correctness specifications and test cases) that describe application specific properties and behaviors provides the basis for developing more effective debugging techniques. We focus on faults in programs that operate on structurally complex inputs, such as heap-allocated data or relational databases. Our approach lays the foundation for a unified framework for localization and repair of faults in programs.

1.3 This dissertation

We embodied our thesis in a suite of integrated techniques based on propositional satisfiability solving, correctness specifications analysis, test spectra analysis, and rule-learning algorithms from machine learning, implemented them as a prototype tool-set, and evaluated them using several subject programs. We describe below some of the key ideas in these techniques for fault-localization and program repair.

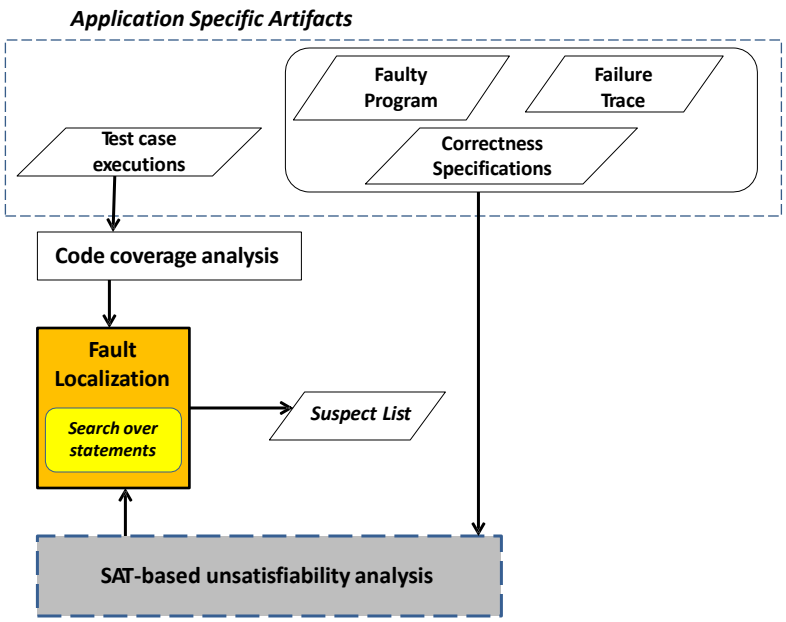

Figure 1.5: Test Spectra analysis + SAT-based Specification analysis for Effective Fault Localization.

1.3.1 Combining SAT-based specification analysis and test-spectra for effective fault localization

We developed a technique that combines spectra-based localization with propositional satisfiability constraint solving in a novel feedback driven approach to localize faults effectively. This technique was published in ASE 2012 [47].

We leverage SAT technology to perform unsatisfiability analysis [101] of the failure trace and the correctness specifications to short-list possibly faulty statements. This information is used to improve the quality of spectra-based localization by i) generation of suitable test cases, and ii) augmenting the ratings of Tarantula, a popular localization tool [64]. Figure 1.5 highlights how this work integrates with our overall solution framework. The specifications of correctness aid in identifying all possible statements responsible for their violation, while analysis of the code

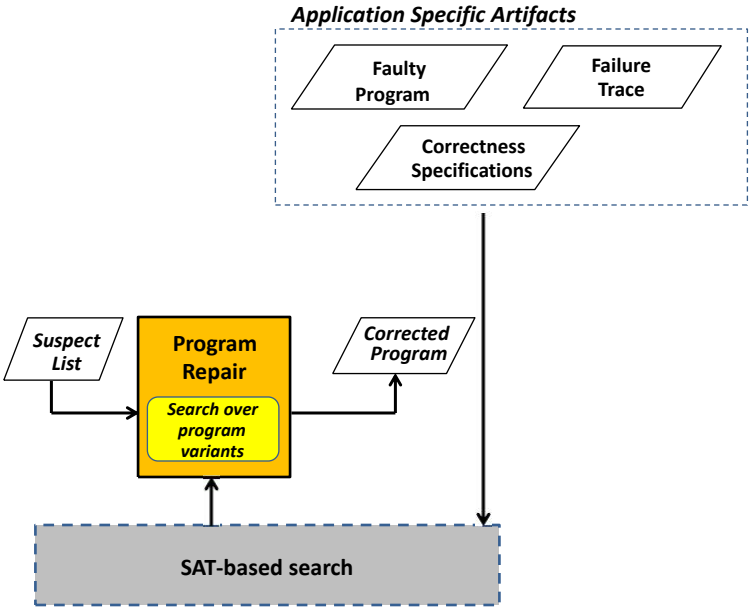

Figure 1.6: Specification analysis + SAT-based search for Effective Program Repair.

coverage of the available passing and failing tests aids in further improving the precision of localization.

Our framework (SAT-TAR) can handle some programs with multiple faults, which is a key challenge for existing techniques. Evaluation of the approach on standard data structure programs and the JTopas parser application shows the technique to be more effective than other state-of-the-art approaches.

1.3.2 Repairing programs with rich correctness specifications

We developed an automated approach for generating likely bug fixes using behavioral specifications. This technique was published in TACAS 2011 [45].

The key idea is to replace a faulty statement that has deterministic behavior with one that has nondeterministic behavior, and to use the specification constraints

to prune the ensuing non-determinism and repair the faulty statement. The state, code and correctness specifications of the procedure are encoded in relational logic, Alloy [59], and then translated down to the boolean satisfiability domain to leverage SAT solvers to systematically search for a solution. Modeling the state of programs as relations enables faster reasoning about properties and operations involving complex state configurations typical of programs dealing with data structures. Initial experiments show the effectiveness of the approach in repairing programs that manipulate structurally complex data with strong specifications of correctness. Figure 1.6 highlights the interaction of the application-specific information in correctness specifications with SAT-based search.

1.3.3 Unified framework for Fault localization and Program Repair

Researchers have traditionally addressed fault localization and program repair as separate problems. We present a technique that performs both these activities in tandem, each providing relevant feedback to the other.

1.3.4 Integration of Machine Learning with Constraint Solving

One of the main challenges in the software debugging domain is the insufficiency of the test oracle problem. The localization of faults and repair of the program are only as accurate as the test oracle. In most practical situations, complete specifications of correctness are not available and the test-suites created by developers mostly do not comprehensively describe the behavior of the program. Machine learning techniques, such as classifier learning algorithms [13, 82, 103],

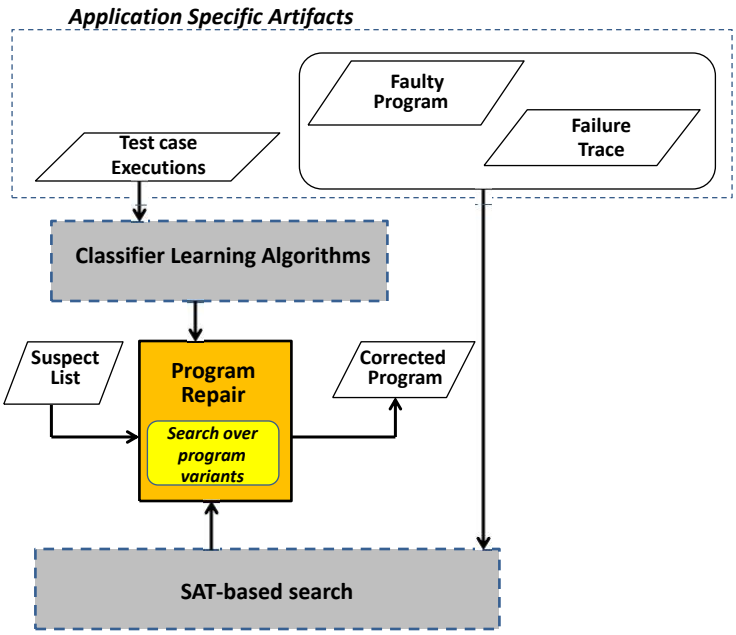

Figure 1.7: Classifier Learning + SAT-based search for Effective Program Repair.

are geared towards towards learning a model of intended behavior based on a set of training data and use the model to predict behavior on unseen future inputs with high accuracy. This is analogous to the problem of synthesizing a repair that works for a given test-suite but is able to generalize to future inputs as well. Our work [43] presents the first use of machine learning techniques in the field of program repair which highlights the potential of machine learning techniques to alleviate the test-oracle insufficiency problem.

Database-centric programs form the backbone of many enterprise systems. Fixing defects in such programs takes much human effort due to the interplay between imperative code and database-centric logic. We presented a novel data-driven approach for automated fixing of bugs in the selection condition of database statements (e.g., WHERE clause of SELECT statements) a common form of bugs in

such programs [43]. Figure 1.7 shows how this work fits within the overall framework of our thesis. This project was done in collaboration with Satish Chandra and DiptiKalyan Saha from IBM Research India, and we published our results at ICSE 2014.

Our key observation is that in real-world data, there is information latent in the distribution of data that can be useful to repair selection conditions efficiently. Given a faulty database program and input data, only a part of which induces the defect, our novelty is in determining the correct behavior for the defect-inducing data by taking advantage of the information revealed by the rest of the data. We accomplish this by employing semi-supervised learning to predict the correct behavior for defect-inducing data and by patching up any inaccuracies in the prediction by a SAT-based combinatorial search. Next, we learn a compact decision tree for the correct behavior, including the correct behavior on the defect-inducing data. This tree suggests a plausible fix to the selection condition. We demonstrate the feasibility of our approach on seven real world examples.

We also extend the idea of applying learning techniques to leverage information from the data-spectra (i.e. distribution of data in the state-space of passing and failing test executions), to correct imperative programs with faulty branch conditions.

1.4 Contributions

This dissertation makes the following contributions:

- **Application of satisfiability solving in the field of debugging.** Considerable amount of work has been done in applying SAT-based analysis for the verification of programs. However, applying it to fault localization and program repair has received lesser attention. We present a novel application of the SAT-based Alloy tool-set to effectively short-list faulty statements using unsatisfiable cores and search for the correct program variant.

- **Leveraging rich specifications of correctness.** Correctness contracts can provide useful insights into the reasons for failure. Existing techniques fail to effectively exploit them. Our support for rich behaviorial specifications (enabled by the Alloy tool-set), aids in improving the precision of fault localization as well as program repair.

- **Handling complex programs and faults.** Most existing debugging techniques are not very effective on data structure programs, owing to their complex structural and behavioral properties. Our approach for repair is novel in its ability to correct data structure programs with complex correctness specifications precisely. Our fault localization approach combines specification-based analysis and the spectra of tests to produce effective localization for different types of faults and user-provided contracts.

- **Dealing with Multiple faults.** We employ specification-analysis based partitioning of failing tests to localize programs with multiple faults. Our repair algorithm is also adept at handling more than one faults in the code.

- **Minimal test-suite generation.** Our framework for fault localization generates the minimum number of test cases required to localize the faults in code.

- **A unified framework for debugging.** We provide a debugging framework which unifies fault localization and program repair while requiring minimal input from the user and automatically makes suggestions for how to fix bugs in a program. Our debugging framework also enables a seamless integration of debugging with existing automated testing and verification tools.

- **Classifier learning to generate repair suggestions.** We describe a new approach for repairing the `where` condition of `select` statement in a database program. The approach is based on the observation that standard decision-tree learning can be used to arrive at a repair suggestion once the *correct* behavior of the `where` is known for the failing keys as well.

- **Combination of machine learning and combinatorics for more effective repair.** We give a new way of combining machine learning and combinatorial search in determining the correct labels for the failing keys. The learning part takes advantage of the known behavior of the passing keys, whereas, the combinatorial part makes up for cases in which the knowledge for passing keys does not extend perfectly to the failing keys.

- **Comparative study with state-of-the-art approaches.** Our evaluation for fault localization highlights some specific issues with localizing faults using purely spectrum-based techniques in the implementations of recursive data

structures. We demonstrate that SAT-TAR localizes faults more accurately than existing state-of-the-art spectra-based and SAT-based localization techniques.

- **Evaluation on real applications.** In addition to performing case-studies on standard data structure programs, we evaluate our approaches for localization and repair on code snippets from the ANTLR [86] and JTopas [1] parser applications. We evaluate the proposed machine learning based repair approach on a excerpts of real ABAP programs and real data drawn from an industrial setting.

- **Machine learning for repairing branch conditions**. We introduce the idea of repairing branch conditions in imperative programs using fundamental tools from machine learning.

- **Prediction of correct behavior**. We define an integrated approach to *efficiently generate the correct behavior* of a branch in an imperative program based on (1) semi-supervised learning to predict expected behavior of failing tests based on similar passing tests, (2) state-space exploration to rectify incorrect predications,

- **Repair suggestions close to oracular fix.** We employ decision tree learning to generate a classifier that represents the repaired condition that is generalizable beyond the existing tests in most cases.

- **Demonstration.** We present experiments to demonstrate the usefulness of

our approach in generating intricate repairs accurately (including synthesis of omitted conditions and correcting loop constructs).

1.5 Organization of Thesis

Chapter 2 discusses background material. Chapter 3 discusses related work. Chapters 4 – 7 provide details of the projects that form the heart of this dissertation. Chapter 8 presents the conclusion.

Chapter 2

Background

This chapter presents a short description of the enabling technology used in our techniques for fault localization and repair. Appendix A provides more details.

2.1 Terminology

The IEEE standard glossary of software engineering terminology [3] defines a *fault* as the manifestation of a human mistake such as using a wrong formula for calculation or de-referencing a null pointer, and a *failure* as the result of the fault such as a wrong value appearing in the output or the program throwing an exception. An *error* is defined as the resulting incorrectness in the program state, for instance, the difference between the expected output value and the actual output of program.

2.2 Use of SAT in software verification

The basic problem that satisfiability solving (SAT) technology seeks to address is as follows, *given a formula in boolean propositional logic, find variable assignments such that the formula evaluates to true, or prove that no such assignments exist.* Recent improvements in rapid search algorithms for SAT [37, 83] have led to its application in a variety of domains such as EDA (Verification, Logic syn-

thesis, FPGA routing), AI (auto theorem proving, KB reduction) and more recently in program verification. Bounded Model Checking [22, 24] leverages SAT technology to verify concurrent, finite state transition systems against properties written in temporal logic. SAT-based predicate abstraction [26] is another area that has received much attention. SATURN is a technique that checks whole programs against properties expressible as finite state machines by encoding the behavior of program modules or procedures in SAT [112].

Jackson and Vaziri introduced the idea of leveraging SAT solvers to perform bounded verification of object-oriented programs against rich properties of correctness. Their approach, embodied in the JAlloy tool [104], comprises of a two-phase reduction process. In the first phase, the code in imperative logic is translated to a formula in relational logic, and in the second-phase the relational logic formula is translated to a problem in boolean satisfiability to leverage off-the-shelf SAT solving technology to look for solutions.

2.3 Relational View of the Heap

The idea that enables expressing imperative code in relational logic is based on the relational view of the heap of a procedure. A problem in relational logic consists of a *universe* which is a finite set of atoms from which solutions to the problem will be drawn, a set of relations on the atoms in the universe, and a set of formulas in which the relations appear as free variables. Hence, the entire state of an object-oriented program is modeled as relations. Java types and classes are represented as sets (relations with arity 1) containing all possible elements of that

```
sig SLL{
    header: lone Node,
    header`: lone Node,
    size: Int,
    size`: Int}
sig Node{
    next: lone Node,
    elt: lone Int,
    next`: lone Node,
    elt`: lone Int}

pred repOk(This: SLL){ //class invariant of SLL
   all n: This.header.*next | n !in n.^next //acyclicity
   # This.header.*next = int This.size //size-invariant
   all n, m: This.header.*next |
      int m.key = int n.key => n = m //unique-elements}

pred deletepostcond(This: SLL, k: Int){
  //remove-ok
  This.header.*next.key - k = This.header`.*next`.key`
  && ((k in (This.header.*next.key))
           => (This.size - 1 = This.size`)) //size-ok}
```

Figure 2.1: `LinkedList` class invariant, post-condition for `delete` in Alloy.

type. Fields are modeled as binary relations mapping elements of the class that they are a member of, to elements of their type. Local variables and method parameters are represented as singleton sets or scalars. The operations in the object-oriented world are mapped to operations on relations. For instance, consider a field dereference such as $x.f$, where x is an object of class A and f is a member of A and is of type B. The corresponding relational logic translation would model f as a binary relation $f : A \rightarrow B$, x as a singleton set of type A, and $x.f$ as the dot product or scalar join of x and relation f. The join would access the element of type B mapped to x by f. Similarly, a field update is encoded as a relational override.

2.3.1 Alloy

Alloy [59] is a specification language that enables modeling and reasoning about object-oriented systems in first-order relational logic. Alloy supports all basic operations on relations and provides constructs such as transitive closure on relations which aid in representing and reasoning about complex properties easily. For instance, Figure 2.1 shows the model of the singly linked list (SLL) class (Chapter 1). The `sig` keyword is used to define signatures of classes such as `SLL` and `Node`. The relation `size` maps elements of `SLL` to the `Integer` set, `header` maps every element of `SLL` to its header `Node`. The multiplicity of a relation is expressed using keywords such as `lone`, `some`, and `no`. For instance, fields such as `header`, and `next` could have their target elements as exactly one node or null, indicated using `lone`.

Given a finite scope, or bound on the number of elements in each of the `sigs` in the model, a constraint-solver such as the *Alloy Analyzer* [59] can be used to systematically check the model against properties expressed as constraints on the model. The constraints are expressed using facts (`fact`), predicates (`pred`), and functions (`func`). Facts are constraints that mandatorily need to be satisfied by the model, while the model can be checked against predicates and functions on a need basis. In the example model, `pred repOk` defines the invariants that the linked list (`This`) needs to satisfy. It consists of three constraints; `acyclicity`, `size-invariant` and `unique-elements`. The `acyclicity` constraint ensures that there is no cycle in the list by checking that no node should be reachable from itself when traversing through the subsequent nodes via the `next` pointer. Operators

* and represent reflexive and transitive closures on relations. The universal quantifier `all` applies the constraint to all elements in the set `This.header.*next`.

The constraints in Alloy are declarative and stateless. In order to distinguish between the pre-state and post-state of a method in imperative logic, additional relations with a back tick ($'$) suffixed to their names are added to represent the respective post-states. The predicate `deletepostcond` (Figure 2.1) specifically checks the functionality of the `delete` procedure and distinguishes between the pre and post states using the back-tick($'$) symbol.

2.3.2 Kodkod

Kodkod [102] is the backend of the Alloy Analyzer. It translates formulas in first-order relational logic to boolean formulas in CNF form, and leverages off-the-shelf SAT technology to find satisfying solutions within given bounds. One of the key features of Kodkod is its support for *partial-instances*. Every relation is declared with a lower and upper bound. The lower bound comprises of tuples that should mandatorily be part of the relation in the generated solution, and the upper bound comprises of tuples that *may* be part of the relation. The lower bound represents fixed parts of the solution (partial instances), which reduces the size of the generated boolean formulas and the search time for solutions.

2.3.3 Forge

The **Forge** [32] tool-set employs Kodkod to verify Java programs against properties of correctness. The basic idea is to obtain a relational logic formula,

$P(s, s')$ corresponding to the Java procedure, that holds true whenever there exists a valid execution starting from the pre-state s and terminates in the post-state s'. Given specifications of correctness $S(s, s')$ in first-order relational logic, a formula of the form $P(s, s') \bigwedge \neg S(s, s')$ is constructed, a satisfying solution to which represents a counter-example to the specification. It represents a valid pre-state or a valid input to the method, that traces a valid path through the code, but yields an output state that does not satisfy the specification.

2.3.3.1 Code to Logic

There are two pre-processing steps that are performed before the translation of code to $P(s, s')$. Firstly, a computation graph of the procedure is constructed. A computation graph is a directed acyclic graph with single entry and exit points. Every loop in the control-flow graph of the procedure is unrolled a pre-defined number of times and is represented as a nested-if statement. Secondly, the code in imperative logic is translated to *Forge Intermediate Language (FIR)*, a simple relational programming language that is amenable to analysis.

Symbolic execution is employed to convert the FIR procedure into relational logic formulas. The state of the symbolic execution engine at each program point has three components; i) *Relational declaration* of the set of relations at that program point (D), ii) *Path Constraints*, set of constraints on the relations that need to be satisfied for execution to reach that program point (P), iii) *Environment* , mapping the variables and fields in the program to the respective relations in logic (E). If there exists a binding of the relations in D to constants such that the constraints

in P get satisfied, then it indicates that there exists an execution that can feasibly reach that program point. E holds the respective program state. The components of the symbolic state is updated after the execution of each statement. The symbolic execution ultimately generates a final symbolic state; D_{final}, including all the declared relations, P_{final}, the path constraints corresponding to all the execution paths in the procedure within the scope of the unrolls, and E_{final}, the final environment. In the specification, $S(s, s')$, s corresponds to the relations in the initial state D_{init}, while s' corresponds to the post-state relations in D_{final}.

Forge looks for a counter-example to the correctness specification $S(s, s')$ using the formulation, $Pre - condition(s) \bigwedge P(s, s') \bigwedge \neg S(s, s')$. Kodkod looks for a binding to all the relations in D_{final} such that; i) the pre-condition specification $Pre - condition(s)$ is satisfied (valid input state), ii) the constraints in P_{final} are satisfied (valid program path) and iii) the negation of the correctness specification $S(s, s')$ is satisfied (invalid output state).

2.4 Machine Learning

In this section, we provide the essential background on the machine learning concepts that we use in this work.

2.4.1 Classifier Learning

Classification is the machine-learning task of assigning distinct classes to data instances, implemented as a classifier. Classification typically involves two steps. The first step to build a model or a *hypothesis* that describes a set of instances,

whose labels or categorization is pre-determined. Such a set is called a *training set* and the instances in it are called *training examples*. The second step is to use the learnt model to predict the labels for unseen data instances or *classify* future data. The setting wherein the labels of all the instances in the training data set are provided up-front is termed as *Supervised Learning*. In *Semi-supervised Learning* some of the training data is unlabeled. When the instances to be labeled are known up-front, instead of adopting an inductive learning approach that infers a generic prediction function, it may suffice to perform accurate labeling for the data at hand. This is called *Transductive Learning*.

Two popular techniques for classifier learning are *decision tree learning* (Section 2.4.2) and *Non-symbolic algorithms* such as *support vector machines* (Section 2.4.2).

2.4.2 Decision Tree Learning

Decision-trees [81] classify instances by sorting them down a tree. Each node of the tree is a test on a feature of the input and each branch represents an outcome of the test which typically is a value that the feature can assume. The leaf nodes represent the different classes. The feature that best divides the instances would be selected as the root node. Once the instances have been categorized based on the root feature, the instances in every category are evaluated again to determine the best splitting criteria. A commonly used evaluation measure to select the feature for splitting is to choose the one that would provide maximum information-gain. *Information-gain* is defined in terms of a measure called *Entropy*, which represents

the impurity in a collection of instances and the amount of information needed to resolve it or classify the instances. These measures are computed as shown below, [82]:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Val(A)} \frac{|S_v|}{S} Entropy(S_v)$$

where

$$Entropy(S) = -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

where, S is the collection of tuples, A is an attribute, $Val(A)$ is the set of values in attribute A, S_v is the subset of S with tuples in which attribute A is v. $p_{\oplus}$ is the fraction of positive samples in the set S, and $p_{\ominus}$ is the fraction of negative samples in the set S.

The classification for an instance is done by tracing a path from the root to the leaf based on its feature values. The entire tree can be expressed as a set of rules that represent the conditions for classification into different classes. Each rule is in the form of disjunction of conjunctions.

In practise, most datasets have continuous real-valued attributes rather than categorical ones. Most algorithms divide the range values into two thresholds and the test has binary outcomes i) value is less than or equal to threshold, and ii) value is greater than threshold. Decision-tree based rule learning algorithms suffer from low accuracy as the dimensionality of the data increases. They also tend to overfit the classifier to the training data, which reduces the generalizability of the rule.

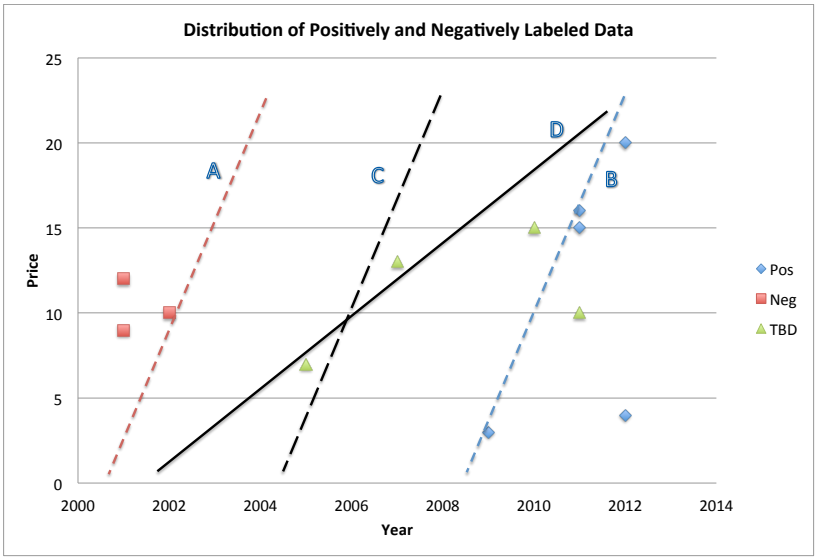

Figure 2.2: Sample distribution of labeled and unlabeled data.

2.4.3 Support Vector Machines

Support Vector Machine (SVM) [13, 103] is a technique based on mathematical optimization. It is based on the principle of finding the *maximum-margin separating hyperplane* that separates positively labeled data from the negatively labeled data. We explain this intuitively using Figure 2.2. Lines labeled A and B both are ”hyperplanes” that separate positively labeled points from the negatively labeled points. Once such as separator is found, an unlabeled point can be assigned the same label as the label of the points on its side of the line.

Neither of the two lines A and B are particularly satisfactory for the purpose of classifying unlabeled points. For example, a point immediately left of line B would be classified negative, which is incorrect. The line C is the one that creates as much margin as possible between lines A and B, and SVMs find such a line. It is expected that such a line would have the best classification behavior, based on what can be inferred from the training data. Lines A and B are called *support*

vectors, which lend their name to the technique. Mathematically, let w be a vector that is normal to the separating hyperplane and let b be its offset. For each positively labeled point x_i, we require $w.x_i \geq b + 1$ and for each negatively labeled point x_i, we require $w.x_i \leq b - 1$. The optimization problem is to minimize $\|w\|$.

SVMs create a classifier based on a *geometric* interpretation of proximity of points in the space. This may not always coincide with the domain-specific measure of proximity (separating line D in Figure 2.2), which is outside the purview of SVM. Fortunately, SVMs provide a numerical measure of prediction confidence. Labels assigned to points close to the separating hyperplane that an SVM discovers (*i.e.* line C in this example) would be assigned a lower confidence, and labels assigned to points far from the separating hyperplane would be assigned a higher confidence. SVMs tend to produce classifiers with high accuracy for linearly separable data. However, there are well known tricks called *kernel methods*, that map data into a higher dimensional space where the data could become separable.

Chapter 3

Related Work

This chapter presents a short description of the existing approaches in fault localization and program repair. More detailed comparison with our work is presented after we describe our techniques in detail.

3.1 Fault Localization

The techniques focusing on fault localization can be broadly categorized as shown below.

Slicing-based approaches: Amongst the oldest approaches in the field of localization are based on the program slicing idea introduced by Weiser [110]. Given a variable with an incorrect value at a program point, *static* slicing uses dependency analysis to determines all statements in the program that might affect the value of the variable at that point. This results in a subset of the program that may contain the fault. However, this subset can be quite large leading to a lot of developer effort in inspection. *Dynamic slicing* [14] uses a test case execution of the program to determine the statements affecting the value of the variable. However, if the precision of the variable deemed suspicious is low, the calculated subset may sometimes miss the actually faulty statement.

Test-spectrum based approaches: A popular method for localizing faults is to analyze the coverage information of the passing and failing tests of the program, also known as the test spectrum. *Set-Union* and *Set-Intersection techniques* fall under this category. Set-Union approach marks as suspicious the set of statements exclusively executed by failing tests and not by any passing tests. The intersection approach on the other hand, determines the set of statements that are executed by every passing test and not by the failing test. The intuition being that the exclusion of these statements might be the reason for the failure. However, these approaches fail when there are passing tests also covering the faulty statement. The *Nearest-Neighbor Queries* technique [90] addresses the issue of the faulty statement being executed by an occasional passing test. Given a failing trace, a passing test is chosen whose coverage is most similar to the failing test and the set of statements executed solely by the failing test are determined. If this initial subset does not contain the fault, the program dependence graph is used to determine the next set of nodes to be examined based on their proximity to the initial set in the graph.

Tarantula [64] is a popular tool that marks a statement as possibly faulty if it is *primarily* executed by failing runs than by passing runs. It associates with each statement a suspiciousness metric that indicates the likelihood of the statement being faulty based on the proportion of failing runs executing it versus passing runs. The localization results are presented in a prototype visualization which enables easy interpretation and inspection. Recent extension to this work [93] applies *Tarantula* on different program entities such as branches, def-use pairs. Abreu et. al introduced another measure of suspiciousness known as the *Ochiai* metric [4],

with its roots in biological study. Elaborate empirical evaluations performed on the Siemens suite [66] indicate *Ochiai* metric to yield the best fault localization effectiveness.

Approaches based on analysis of state and contracts: Zhang et al. [114] and Huang et al. [57], perform analysis of program states rather than locations to identify infected states, leading to more precise localization. BugAssist is a recently developed tool [67], which employs SAT-based analysis of user-provided correctness contracts to determine the portion of code responsible for their violations. Model checking techniques [25] which produce a counter-example or a sequence of state-transitions corresponding to the failure have also been employed to locate faulty state transitions. Ball et al. [10] introduced an approach that compares the counter-examples returned by the model checker with successful traces to identify faulty state transitions.

Divide-Conquer based approaches: Another popular methodology for localization is to apply a *Divide and Conquer* technique to narrow down on the set of computations or state variables that may be causing the failure. The approach proposed by Shapiro and Renner works on this idea by recursively dividing the computation tree of a program into sub-trees [96]. Each sub-tree corresponds to a sub-computation, the output of which needs to be validated. The sub-trees whose output is incorrect are locations of faults. Cause-Transitions technique [27], belonging to the Zeller's suite of **Delta-debugging** techniques, uses the divide and conquer idea to map failures to subsets of state variables. A binary search on the memory states between a passing and failing test is performed to determine a minimal set caus-

ing the failure. A symbolic debugger is used to compare and swap memory states between the two runs at desired program points to observe the effect on the output.

3.2 Program Repair

The problem of program repair has been the focus of a number of techniques, including those based on evolutionary algorithms [109], program code transformations [29], as well as program state mutations [18].

Mutation-based approach: Debroy et al. [29] introduced the idea of using *mutations*, i.e., syntactic transformations to the faulty program as a basis of repair. They developed their technique in the context of the Tarantula tool and spectrum-based fault localization using a given set of passing and failing tests to focus mutations. While such code transformations can assist in debugging, the space of variants to explore grows very quickly. The feasibility of using such a technique for real applications requires developing novel pruning strategies.

Search over syntactic transformations of the program: Weimer et al. [109] introduced the idea of program repair using genetic programming, where existing parts of code are used to patch faults in other parts of code and patching is restricted to those parts that are relevant to the fault. Ackling et al. [5] repair a program by evolving patches to fix it rather than evolving the faulty program itself, and argue that doing so simplifies the repair problem. Wilkerson et al. [111] present a co-evolutionary approach where code and its tests are co-evolved to improve the bug finding ability of tests as well as to improve the overall quality of the code in order to provide an automated software correction system.

Search in the program state space: Chandra et al. [18] use changes to program states in a faulty program to approximate the behavior of a correct program with respect to a given set of passing and failing tests. Sem-Fix [85] is a recent approach that builds on the same idea. Given suite of failing and passing tests, it uses concolic execution to determine a correctness constraint at the location of the fault . It then employs component based synthesis to synthesize an expression that satisfies this correctness constraint. The oracle at the point of fault, reduces the search space to relevant expressions that would make the tests pass. NOPOL [30] and DirectFix [80] are successors Sem-Fix that extend the approach to different fault classes.

Malik et al. [77] use a search-based technique for data structure repair [68] as a basis of program repair. Specifically, they use mutations done on program state to fix corrupt data structures as a basis of synthesizing program statements that abstract those fixes using program variables. Jobstmann et al. [63] introduced a technique to replace faulty program expressions with unknowns and formed a model checking problem in order to repair a faulty program with respect to its linear time logic specification. Griesmayer et al. [50] map the problem of repairing boolean programs to finding a memoryless, stackless strategy in a game and explore the game graph to find a repair for the boolean program, and show how it can be used to repair a class of C programs. Weimer [107] proposed an approach that uses counter-examples of safety-policy violations generated by model checking tools, to generate patches to repair the model. The patches are in the form of sequence of atomic reads and updates that generate a state satisfying the policy.

Combination of search in program state and mutations space: Staged Program Repair (SPR) [76] is a recently proposed approach that generates a fixed set of mutation templates for the suspicious statements. For condition templates , it first determines the correct truth sequence using an efficient technique based on heuristics instead of symbolic execution. The synthesis effort is reduced to instantiation of the template parameters instead of synthesis from scratch.

Combination of specifications and test cases: Wei et al. [106] attempt to combine specification-based and test-based repair. Boolean queries are used to build an abstraction of the state, which forms the basis to represent contracts of the class, fault profile of failing tests and a behavioral model based on passing tests. A comparison between failing and passing profiles is performed for fault localization and a subsequent program synthesis effort generates the repaired statements. This technique however only corrects violations of simple assertions, which can be formulated using boolean methods already present in the class.

CodeHint [40] dynamically generates a code snippet at a particular location for a particular input. It offers support to accommodate different synthesis techniques and different levels of specifications.

Program synthesis. A closely related area to program repair is *program synthesis* [53], where the goal is to generate (parts of) a program independently of a given incorrect version. A number of program synthesis techniques are based on specifications. Programming by sketching [98] employs SAT solvers to generate missing parts of a given skeletal program with respect to another reference program that serves as a specification. A SAT solver completes the implementation details by

generating expressions to fill the "holes" of the skeletal program by exploring several of its variants. Gulwani et al. [53] use the counterexample guided iterative synthesis paradigm together with SMT solvers to synthesize loop-free programs with respect to given specifications of desired functionality. Kuncak et al. [70] generalize decision procedures into synthesis procedures to synthesize code snippets from specifications.

To alleviate the burden of writing detailed specifications, some recent techniques support synthesis based on given concrete input/output examples. Gulwani [52] presents such a technique for synthesizing string processing code for spreadsheets using examples of how a user processes sample strings. More recently, Singh et al. [97] integrate *scenarios*, which illustrate steps of modifying specific data structure instances, with given code skeletons and inductive definitions to facilitate program synthesis.

3.3 Machine Learning in Debugging

The application of machine learning to debugging is largely confined to fault localization. Decision tree generation algorithms, have been used in conjunction with the fault localization tool Tarantula to cluster failing tests in order to help developers manually fix bugs in their code more effectively [17, 65].

Statistical debugging techniques [60, 73] employ statistical analysis on the data collected from passing and failing program runs to determine likely faulty statements. The technique instruments the program with boolean predicates at different points. The boolean vectors yielded by the passing and failing tests are sub-

jected to Logical Regression based classification. A ranking is performed based on the co-efficient of the regression and the predicate along with the respective program point with the highest co-efficient value is likely to be faulty. SOBER [75] is another technique which selects the instrumented predicate with the highest correlation with failure. It specifically chooses only those predicates whose evaluation patterns in the failing runs is significantly different from the passing runs. A number of recent techniques analyze the statistical correlations between test failures and control-flow predicates [73, 75], path profiles [21], and changes made by the programmer [89].

Context-Aware Debugging [61] employ statistical classification using support vector machines and random forests for determining likely faulty statements. Most recently, Roychowdhury et al. [91, 92] used latent divergences and the RELIEF algorithm for feature selection in fault localization.

Program Repair using Machine Learning.

In [6], Alijareh et.al. describe an approach that employs logic learning to diagnose and correct errors in finite state transition systems. Given a set of positive and negative examples, inductive logic programming (ILP), is used to derive hypothesized logic program which entails all the positive and none of the negative examples.

The work presented in [31], employs the Daikon tool to learn specifications for data-structure programs. The specification learnt is used to perform data-structure repair for the failing tests.

Chapter 4

SAT based Fault Localization

This chapter is based on our paper, "*Improving the effectiveness of spectra-based fault localization using specifications*", published at the 27^{th} IEEE/ACM International Conference on Automated Software Engineering (ASE 2012) [47]. Dr. Nokhbeh Zaeem and Dr. Khurshid were the co-authors on this paper. Dr. Zaeem helped me with the experiments and writing some sections of the paper. Dr. Khurshid helped me brainstorm the idea and also contributed on many sections in the paper.

4.1 Summary

We present a technique, **SAT-TAR**, that combines specification-based analysis with dynamic test execution information to produce effective localization consistently across varying fault and specification complexity [47]. Our insight is that analysis of correctness specifications can guide the search for the faulty statements by discerning the actual reason for failure. This can potentially yield higher localization accuracy than existing techniques that employ pure test coverage based analysis [64]. We leverage SAT to perform unsatisfiability analysis of the specifications along with the failure trace to determine a *Minimal Unsatisfiable Core (MUC)* that

consists of the violated specification constraint and the slice of the trace responsible for the violation. This lends more context to aid in further investigation as against an arbitrary ranking of unrelated statements as done by existing approaches.

Complete reliance on user-defined contracts is also problematic since the completeness and complexity of the constraints can adversely impact the time required to effectively locate the faults [11, 51, 67]. Application of specification and test spectra-based analysis in tandem would produce more precise results in lesser amount of time than pure specification based analysis.

```
class LinkedList{
      Node header;
      int size; // number of nodes in this list
      static class Node{
         Node next;
         int key;}
      boolean delete(int k){
         // removes the node with key value k
 0:      Node prev = null;
 1:      Node lst = header;
 2:      while(lst != null){
 3:         if(lst.key == k){
 4:            if(prev != null){
 5:               prev.next = lst.next; //correct
                    prev.next = lst; //removeErr
               }else{
 6:               header = header.next; //correct
                    header = header; //headErr
               }
 7:            size--; //correct
                size = 0; //sizeErr
 8:            return true;}
 9:         prev = lst;
10:         lst = lst.next;}
11:      return false;}}
```

Listing 4.1: `LinkedList` and its `delete` method with faults.

Our approach specifically generates tests that would provide effective localization for the fault causing the violation. We enable incremental generation and processing of tests. Heuristic analysis of the ratings after every round is used to

refine the localization and provide *feedback* to improve the test generation strategy on-the-fly. Test selection strategies face the challenge to balance between choosing sufficient number and type of tests and keeping the test-suite size manageable under time constraints. SAT-TAR generates test-suites containing *minimal number of test cases* required to produce high quality localization.

Our technique provides effective localization even in *multiple fault scenarios*. We perform root-cause analysis of the specification violations and support user-control on addition of tests to prevent different faults from interfering with each other's localization.

4.2 Illustrative Overview

We consider the localization of typical faults in the linked list `delete` method (Listing 4.1) that we had introduced earlier in Chapter 1. We first highlight the demerits of applying existing state-of-the-art approaches to locate the faults; **pure spectra-based localization** (Tarantula (**TAR**)) on a test-suite generated using the most recent *directed test generation* technique for effective fault localization, (**DT**) [8], and **pure SAT-based analysis of specifications (SAT)**. We next illustrate the application of our approach, **SAT-TAR**, to locate the same faults.

Example program with faults: Let us consider two faulty versions of the `delete` method, one comprising of just `sizeErr` on statement 7 and the other comprising of just `headErr` on statement 6. The first fault wrongly sets the `size` field to 0 instead of decrementing it whenever a match is found. The second fault incorrectly updates the `header` field. We consider two possible versions for the

```
pred repOk(This: LinkedList){ // class invariant
   all n:This.header.*next | n !in n.^next//acyclicity
   # This.header.*next = int This.size//size-invariant
   all n, m: This.header.*next |
      int m.key = int n.key => n = m//unique-elements}

pred deletepostcond1(This: LinkedList, k:
Int){
  //remove-ok
  This.header.*next.key-k=This.header`.*next`.key`
  && ((k in (This.header.*next.key))
             => (This.size-1=This.size`))//size-ok}

pred deletepostcond2(This: LinkedList, k: Int){
   repOk[This]}
```

Figure 4.1: Two different post-conditions for the `delete` method.

user-defined post-condition specification for the `delete` method (Figure 4.1). In `deletepostcond1`, the user includes constraints to specifically check the functionality of the method alone; only the node with the matching key value has been removed from the list (*remove-ok*) and the size of the list has been decremented on a successful delete (*size-ok*). On the other hand, in `deletepostcond2`, the user ensures that all the class invariants are preserved after deletion without including a specific check to ensure the removal of the input value. The class invariant (`repOk`) consists of the *acyclicity* constraint, which checks that the list does not contain any cycles, the *unique-elements* constraint, which checks that each list node has a unique key, and *size-invariant*, which ensures that the `size` field has a value equal to the number of nodes reachable from the `header` through the `next` pointer. We use these two versions to show how our technique is intended to work with varying levels of complexity and completeness in user-defined specifications.

Applying Tarantula (TAR):

The **Tarantula** tool (Chapter 3) localizes faults in a program based on the

Table 4.1: Spectra-based localization using Tarantula.

Stmts				0	1	2	3	4	5	**6**	**7**	8	9	10	11
Fault	Tech.	Tests	Rating												
sizeErr	Tar^{DT}	2F,2P	Susp $_{Tar}$	0.5	0.5	0.5	0.5	0.6	1.0	0.5	**0.6**	0.6	0.5	0.5	0.0
			Conf $_{Tar}$	1.0	1.0	1.0	1.0	1.0	0.5	0.5	**1.0**	1.0	0.5	0.5	0.5
			Rank	8	8	8	8	4	1	8	**4**	4	10	10	11
headErr	Tar^{DT}	2F,2P	Susp $_{Tar}$	0.5	0.5	0.5	0.5	0.6	0.0	**1.0**	0.6	0.6	0.0	0.0	0.0
			Conf $_{Tar}$	1.0	1.0	1.0	1.0	1.0	0.5	**1.0**	1.0	1.0	1.0	1.0	0.5
			Rank	8	8	8	8	4	12	**1**	4	4	10	10	12

coverage of a given test suite. It associates with each statement a metric of *suspiciousness*, which represents the likelihood of the statement being faulty, and a metric of *confidence* in the calculated suspiciousness. In the following definitions, $Failed(s)$ and $Passed(s)$ show the number of failing and passing tests covering statement s respectively, while $TotalFailed$ and $TotalPassed$ represent the total number of failing and passing tests.

$$Susp_{Tar}(s) = \frac{\frac{Failed(s)}{TotalFailed}}{\frac{Failed(s)}{TotalFailed} + \frac{Passed(s)}{TotalPassed}}$$

$$Conf_{Tar}(s) = Max(\frac{Failed(s)}{TotalFailed}, \frac{Passed(s)}{TotalPassed})$$

The values of the both the metrics range from 0.0 to 1.0. Statements with a suspiciousness value of 1.0 are considered to be the most suspicious. Among the statements with the same suspiciousness values, those having a higher value for confidence are more likely to be faulty since more number of tests lend evidence to them being suspicious. A *suspect list* of statements is produced, which enlists the statements in descending order of the suspiciousness and confidence metrics. Each statement is assigned a **rank**, which indicates *the maximum number of statements*

that would need to be examined to reach that statement (inclusive of the statement), if they were to be arranged in the descending order of suspiciousness and confidence values. The worst-case rank assigned to the faulty statements represents the effectiveness of the localization.

We first apply Tarantula on a minimal test suite. Consider the use of just two tests; a failing test case with a two node input list and the value `k` matching the `key` of the second node, and a passing test case with an empty input list, to localize `sizeErr`. The traces of the two runs are very dissimilar, so most of the statements are assigned the same suspiciousness and confidence ratings of 1.0. Hence in the worst case 6 actually correct statements should be examined before hitting the faulty statement 7, giving it a **rank** of 7.

The results improve on using a test-suite with 4 tests generated specifically for effective fault localization (Tar^{DT} in Table 4.1). While `headErr` is well-localized ranking the faulty statement 6 first in the list, the ratings are not accurate for `sizeErr`. Statement 7 (rank 4) is rated lower than the actually correct statement 5. This is because the suite includes a test that removes from an input list with just one node, for which setting size to 0 after deletion is valid. Hence statement 7 has one passing test covering it, whereas all tests covering statement 5 are failing runs.

Applying pure SAT-based analysis of specifications (SAT): Let us now consider the application of a technique which solely uses static analysis of the code and violated specifications, such as BugAssist, to localize the same faults.

The BugAssist tool (Chapter 3) employs user-provided specifications of cor-

rectness to determine the portion of code responsible for the violations. A problem is formulated in boolean logic and SAT technology is employed to look for a satisfying solution that represents a valid path through the code that produces an output satisfying the correctness specifications for the given input. However, due to the fault in the code this formula is unsatisfiable. A MAX-SAT solver [79] is used to determine a *maximal* set of clauses that can be simultaneously satisfied by any assignment. A complement of these clauses represents a *Minimal Unsatisfiable Set*. The replacement of some or all of the clauses in this set with a different formula should yield satisfiability. These are mapped to corresponding source code statements which are considered to be potentially faulty. There may be more than one such minimal unsatisfiable sets. Hence, if a particular set does not map to the actual fault then repetitive calls to SAT are performed to parse through all possible faulty statements.

Consider localizing `headErr` using `deletepostcond2` as the post-condition specification. A list with the key value of the first node matching the input `k`, would expose this fault. The violated constraint is `size-invariant`, which includes almost all fields of the linked list (`size`, `header`, `next`). Localization based on analysis of specifications and code would include all statements that impact the values of these relations in their post-state (statements 0,1,2,3,4,6,7).

However, complex specifications, made up of a number of dependent constraints, result in multiple violated constraints involving many data structure fields.

.

Applying SAT-TAR: The above experiments lend motivation to our intuition that

Table 4.2: Pure SAT and SAT-TAR localizations for `sizeErr`.

Stmt	*SAT*		*Spectra* Tar^{MUC}						
	postcond 1		1F,1F		2F,1P		2F,2P		
	Susp $_{SAT}$	Rank $_{SAT}$	Susp_1	Conf_1	Susp_2	Conf_2	Susp $_{sattar}$	Conf $_{sattar}$	Rank $_{sattar}$
0	0.0	12	1.0	1.0	0.5	1.0	0.5	1.0	7
1	1.0	4	2.0	1.0	1.5	1.0	1.5	1.0	4
2	1.0	4	2.0	1.0	1.5	1.0	1.5	1.0	4
3	1.0	4	2.0	1.0	1.5	1.0	1.6	1.0	2
4	0.0	12	1.0	1.0	1.0	1.0	1.0	1.0	6
5	0.0	12	1.0	0.5	1.0	0.5	1.0	0.5	9
6	0.0	12	1.0	0.5	1.0	0.5	1.0	0.5	9
7	**1.0**	**4**	**2.0**	**1.0**	**2.0**	**1.0**	**2.0**	**1.0**	**1**
8	0.0	12	1.0	1.0	1.0	1.0	1.0	1.0	6
9	0.0	12	1.0	0.5	0.3	1.0	0.5	0.5	9
10	0.0	12	1.0	0.5	0.3	1.0	0.5	0.5	9
11	0.0	12	0.0	0.0	0.0	1.0	0.0	1.0	10

Table 4.3: Pure SAT and SAT-TAR localizations for `headErr`.

Stmt	*SAT*		*Spectra* Tar^{MUC}						
	postcond 1		1F,1P		2F,1P		2F,2P		
	Susp $_{SAT}$	Rank $_{SAT}$	Susp_1	Conf_1	Susp_2	Conf_2	Susp $_{sattar}$	Conf $_{sattar}$	Rank $_{sattar}$
0	1.0	7	1.5	1.0	1.5	1.0	1.5	1.0	7
1	1.0	7	1.5	1.0	1.5	1.0	1.5	1.0	7
2	1.0	7	1.5	1.0	1.5	1.0	1.5	1.0	7
3	1.0	7	1.5	1.0	1.5	1.0	1.6	1.0	4
4	1.0	7	1.5	1.0	1.6	1.0	1.75	1.0	3
5	0.0	12	0.0	1.0	0.0	0.5	0.0	0.3	12
6	**1.0**	**7**	**2.0**	**1.0**	**2.0**	**1.0**	**2.0**	**1.0**	**1**
7	1.0	7	1.5	1.0	1.6	1.0	1.75	1.0	3
8	0.0	12	0.5	1.0	0.6	1.0	0.75	1.0	8
9	0.0	12	0.0	1.0	0.0	1.0	0.0	0.6	11
10	0.0	12	0.0	1.0	0.0	1.0	0.0	0.6	11
11	0.0	12	0.0	0.0	0.0	0.5	0.0	0.6	11

instead of dwelling just in the realm of pure spectra-based localization or pure SAT-based analysis, it would be helpful to combine the two techniques in order to achieve effective localization consistently across different fault scenarios.

We present below the main steps of our approach.

SAT-based localization: Given a program annotated with correctness specifications and the trace of a single run violating these constraints, our approach investigates the violated constraints to determine the list of statements responsible for the specification violation. We employ the Forge framework, with Kodkod at its backend, to determine a Minimal Unsatisfiable Core of the violated specification constraints. The MUC consists of the core constraints responsible for the violation and is mapped back to an initial suspect list of potentially faulty source code statements. The specification constraint violated by the input exposing the `sizeErr` is the `size-ok` constraint. The suspect list produced using unsatisfiability analysis includes the faulty statement 7 updating `size` while eliminating statements updating other fields such as statement 5. Faulty statement 7 (Rank_{SAT} in Table 4.2) is ranked fourth using just a single failing test.

However, complex or incomplete user-defined specifications can yield long suspect lists such as in `headErr`. In situations unsatisfiability analysis based filtering does not help much in narrowing down on the faulty statements (Rank_{SAT} in Table 4.3).

Test Spectra-based localization: Instead of performing further analysis in the SAT-domain, SAT-TAR performs test spectra-based refinement which is more efficient. We use the test-coverage based metrics of the popular *Tarantula* tool [64] to obtain suspiciousness ratings for the statements.

We utilize the information obtained from the UNSAT core analysis in two ways. Firstly, we generate the test cases for spectra-based localization using a variant of the directed test generation approach [8]. Specifically, we generate the test

input whose trace is most similar to the failure trace in terms of its coverage of *only those statements that appear in the MUC* and hence contributes to the violation. Spectra-based localization using test cases thus generated, results in more accurate ratings using fewer number of tests. Secondly, we increase the suspiciousness ratings of those statements that appear in the MUC. Specifically, we add a value of 1.0 to the suspiciousness of statements in the suspect list built using the MUC analysis (Tar^{MUC} in Tables 4.2 and 4.3)

Instead of generating a whole test-suite upfront, we perform incremental generation and processing of tests. The tables show the tests incrementally added to the test-suite and the corresponding ratings. For `headErr`, the test case added in the first round is a passing run wherein the input `k` matches the key of the second node of the list. This test has its code coverage most similar to the failing run, hence the suspiciousness of all statements in common between the two traces gets reduced contributing to significant refinement in the ratings (Susp_1 and Conf_1 in Table 4.3). Such a passing test case is generated after the second round for `sizeErr` (Susp_2 and Conf_2 in Table 4.2). This process continues until no more tests with unique coverage of the code can be generated.

Incremental processing of test cases facilitates the availability of intermediate localization results which is beneficial, as elaborated in Section 4.3. Particularly, the user is presented with the results after every round and he could choose to stop the addition of tests and further processing when appropriate. In both the fault cases, the best possible localization of the faulty statements is achieved using the first few tests. Hence, ideally the user could stop further processing after the second

Table 4.4: Multi-fault Localization.

Faulty Stmt	Tar^{DT}			SAT-TAR(postcond1)					
	test suite shows both faults			initial trace shows **headErr**			initial trace shows **sizeErr**		
	3F,3P			1F,2P			1F,2P		
	Susp Tar	Conf Tar	Rank Tar	Susp $sattar$	Conf $sattar$	Rank $sattar$	Susp $sattar$	Conf $sattar$	Rank $sattar$
6	1.0	0.6	**4**	**2.0**	**1.0**	**2**	0.5	1.0	11
7	1.0	1.0	**3**	1.0	1.0	8	**2.0**	**1.0**	**1**

round for `sizeErr` and the first for `headErr`.

Observe that faulty statements 7 and 6 are both assigned suspiciousness of 2.0 and confidence of 1.0 respectively (Susp$_{sattar}$ and Conf$_{sattar}$ in Tables 4.2 and 4.3), which are higher than the ratings for the correct statements. Hence they rank first in a descending ordering of the statements based on their suspiciousness and confidence. Compared to the counterpart techniques (TAR, SAT), we provide high quality localization consistently for both the fault scenarios.

Multiple Faults: In multi-fault cases, tests covering different faults interfere with each other impacting the effectiveness of pure spectra-based localization. We applied Tarantula on a code containing both the faults (`sizeErr` and `headErr`). The test suite was generated using the directed test generation approach for effective fault localization (*DT*) and had failing tests covering both of the faults. The localization (Tar^{DT}) did not prove very effective as the faulty statements 7 and 6 ranked third and fourth respectively (Table 4.4).

Our approach targets localizing only the faults exposed by the initial failing trace, and performs violated-specification based partitioning of the suite to eliminate failing tests covering different faults. The smallest fault revealing input for

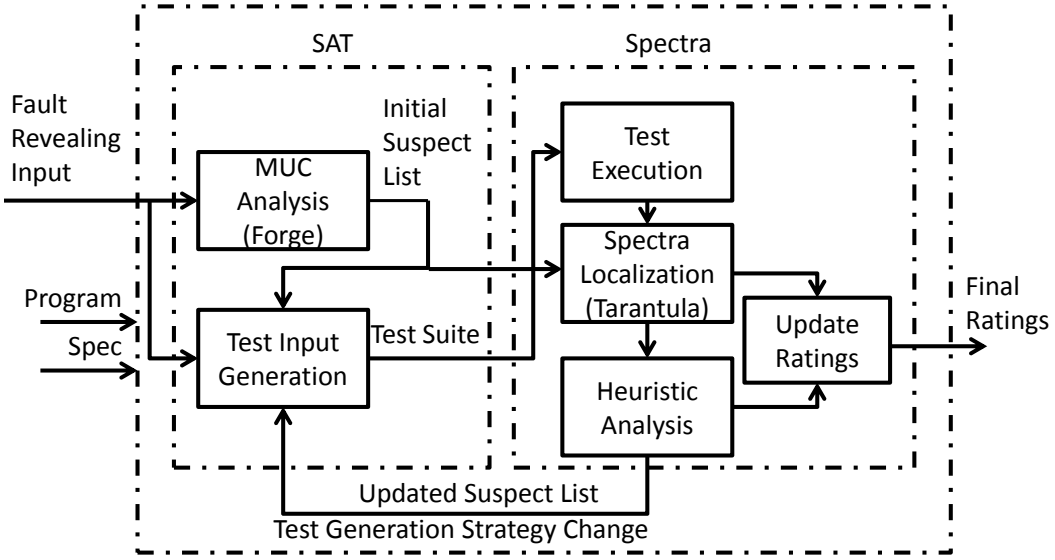

Figure 4.2: SAT-TAR framework

the two fault example would be a run removing a node from a list with only one node. This run violates the `remove-ok` constraint alone caused due to `headErr` on statement 6. Failing runs covering `sizeErr` and violating `size-ok` constraint are not included in the test-suite, thus preventing interference with the localization of `headErr`. The suspect list built using MUC analysis, also aids in augmenting the suspiciousness of statement 6, ranking it second in the list. Similarly when starting with an input violating `size-ok` alone, our approach eliminates the failing runs violating the `remove-ok` constraint, thus localizing statement 7 with high precision with a rank of 1 (`sizeErr` in Table 4.4).

4.3 Framework Details

In this section we elaborate on the algorithm and implementation details of the modules of our framework. Figure 4.2 presents an overview. The input to the framework is a fault-revealing test which can be produced by any testing or static verification technique [16, 20]. The two main constituents of our framework are SAT-based analysis (Section 4.3.1) and spectra-based analysis (Section 4.3.2).

4.3.1 SAT

Minimal Unsatisfiable Core Extraction and Analysis:

Considerable work has been done in recent years to find minimal unsatisfiable cores of unsatisfiable constraints written as propositional satisfiability (SAT) formulas [35, 113]. Given an unsatisfiable CNF formula X, a minimal unsatisfiable sub-formula (MUS) is a subset of X's clauses that is both unsatisfiable and minimal, which means any subset of an MUS is satisfiable. There could be many independent reasons for a formula's unsatisfiability and hence more than one minimal cores. Some algorithms such as CAMUS [74] aim to extract all the MUS's at the cost of not being able to scale to problems of real world sizes. Additionally, when using boolean logic to reason about specifications in relational logic, mapping of unsatisfiable cores in boolean logic to equivalent relational logic constraints becomes complex.

Torlak et al. [101] propose an efficient algorithm for the extraction of a single MUC of declarative specifications based on the resolution refutation proofs generated by SAT solvers and theorem provers. The Recycling Core Extractor al-

gorithm (RCE), returns an unsatisfiable core of specifications written in the Alloy language that is guaranteed to be sound (constraints not included in the UNSAT core are definitely irrelevant to the unsatisfiability proof) and irreducible (removal of any constraint from the set would make the remaining formula satisfiable). The two key ideas of this strategy are i) finding the minimal core at the specification level rather than at the boolean level by iterative application of reduction and translation, and ii) enabling reuse of the results of inferences made in earlier iterations to increase performance. MUC has been shown to be useful in the identification of over-constrained models, weak theorems, and insufficient scopes while checking models. We extend its use MUC in localizing faults in imperative code.

```
Trace testGeneration(List<Stmt> suspectList,Trace ce,List<Trace> selTraces,
    TestGenStrategy st) {
  Constraint[] pathConds = pathCondConstraints(ce);
  for(int i = 0; i < pathConds.length; i++) {
     Constraint newPC=conjunction(pathConds[0],...,pathConds[i-1],negation(
         pathConds[i]));
      Trace newTrace = KodKod.solve(conjunction(preCond,newPC));
      if (newTrace.satisfiable()) {
        if (st.equals(NEW_STRATEGY))
           boolean uniqueCov = uniqueCoverage(newTrace, selTraces,suspectList);
        else
           uniqueCov=uniqueCoverage(newTrace,selTraces);
        if (uniqueCov)
           List<Trace> newTraces.add(newTrace);}}
  int max = 0;
  Trace similarTrace = null;
  for(Trace t : newTraces) {
     List<Stmt> stmtNums = srcCodeStmts(t);
     for(Stmt st : suspectList) {
        if (stmtNums.contains(st)) score++;
        if (max < score)
           {max = score; similarTrace = t;}}}
   selTraces.add(similarTrace);
  return similarTrace;}
```

Listing 4.2: Test Input Generation Algorithm.

The Forge framework (Chapter 2) uses symbolic execution to translate the code of a procedure into a set of formulas in relational logic ($F_{code} = \{f_1, ..., f_n\}$).

Given a fault-revealing input (I_{fr}) and the correctness specification (ψ), we attempt to find a valid path through the code for the given input, yielding an output satisfying the post-condition specification. Kodkod is invoked with the following Alloy formula; $F_{code} \wedge \psi$. The pre-state relations are bound to the values in the fault revealing input (I_{fr}) and the scope is fixed to the size of the input. Due to the fault in the code, the formula is unsatisfiable and Kodkod returns a proof of unsatisfiability (UNSAT). The RCE strategy of Kodkod is used to produce the MUC, a reduced form which consists of the core constraints responsible for the violation. $MUC = \{\{f - muc_1, ..., f - muc_m\}, \{\psi - muc_1, ..., \psi - muc_m\}\}$, which is some subset of $\{\{f_1, ..., f_n\}, \psi\}$.

Figure 4.3 shows the main formulas that appear in the MUC extracted for the `sizeErr`. The first formula represents the violated specification constraint `size-ok`. The second formula represents the constraints that appear on the path that updates the size field. The first two constraints represent conditionals on the pre-state relations evaluated by branches 2 and 3 (Listing 4.1). The last constraint represents the faulty statement 7, as an override of the size relation wherein the list instance, `This`, is mapped to a value 0. Note that only those branches that are relevant to the violation (branches on which the update statement is control-dependent on), are included in the MUC. For instance, branch statement 4 is not included in the MUC for `sizeErr`.

Forge maintains a formula slice map when performing the translation of imperative code to logic. This maps each formula inserted into the path-constraint (f_i), to the set of statements or slice of code from which the formula was gener-

```
((k in (This.header.*next.key)) =>(This.size-1=This.size`))

!(no  This.header)=> (This.header.key = val)=>
This.size` = size ++ (This -> 0)
```

Figure 4.3: MUC for `sizeErr`.

ated [34]. This is utilized to determine the set of statements mapping to the formulas in the MUC ($f - muc_i$), which can be considered responsible for the unsatisfiability. This forms the initial list of suspicious statements. For the `sizeErr`, statements 1,2,3,7 are short-listed using this analysis from the list of all statements in the trace 0,1,2,3,4,6,7,8. The suspiciousness of statement s is, **Susp$_{SAT}$(s) = 1.0**, if statement s is in the initial suspect list, otherwise it is 0.0.

Kodkod returns only a single MUC for a violation, which is not sufficient when the trace covers multiple faults violating more than one constraints. Hence after processing an MUC, we remove the corresponding violated constraints from the specifications, and invoke the unsatisfiability analysis again to extract other MUCs. We thus form a consolidated list of suspicious statements covering all faults in the code.

Test Input Generation:

This module aims to generate test inputs that would provide the most benefit in localizing the fault exposed by the initial failure trace. The algorithm involves two main steps (pseudo-code in Listing 4.2): i) using the conditionals on the failure trace to generate a set of additional inputs covering new portions of the code, and ii) selecting the input which is the most *similar* to the failure trace as the next test input (`similarTrace`). The heuristic behind this technique is that if the input with a trace most similar to the failure trace happens to be a passing run then it

would provide the most benefit in improving the precision of the localization, since a considerable number of statements in common between the two traces could be considered non-faulty. The first step is accomplished by systematically negating each of the path constraints on the failure trace, $\{f - pc_1, ..., f - pc_n\}$, obtained by solving for F_{code} binding the pre-state to I_{fr}. The following new path constraints get generated; $f - pc_1 \wedge f - pc_2 \wedge ... \wedge \neg f - pc_n$, $f - pc_1 \wedge f - pc_2 \wedge ... \wedge \neg f - pc_{n-1}$ etc. until $\neg f - pc_1$. For every new path constraint (`newPC`), SAT is employed to look for a valid input whose trace includes that constraint.

Given a failure trace and its unsatisfiability core (MUC), we define the *similarity score of another trace with respect to the failure trace, as the number of source code statements covered by it that map to the MUC.* The trace with the highest similarity score is considered the most similar. The selected trace needs to cover at least one statement from the suspect list. Such a trace would provide the most benefit in refining the suspect list generated using MUC analysis. The most similar test input along with the initial fault-revealing input form the initial test-suite.

4.3.2 Spectra

The tests generated in the SAT domain are executed, followed by spectra-based localization using the formulas of Tarantula. The suspiciousness and confidence metrics are calculated as follows,

Susp$_{sattar}$**(s) = Susp**$^{MUC}_{Tar}$**(s) + Susp**$_{SAT}$**(s)**

Conf$_{sattar}$**(s) = Conf**$^{MUC}_{Tar}$**(s).**

The superscript ^{MUC} represents MUC based test generation.$_{Tar}$ indicates Tarantula's spectra-based localization metrics.$_{SAT}$ represents the metric obtained from SAT-based analysis.

Violated specification based filtering of tests: For failing tests, the violated constraints are analyzed to confirm that they match the constraints violated by the initial failing run. Tests that violate distinctly different constraints are not considered since they pass through different code faults and would wrongly reduce the ratings of the faulty statements in the initial failure trace.

Heuristic Analysis of Ratings: The test inputs are generated and used for localization one at a time. Such incremental addition and processing of tests facilitates application of heuristics to refine the precision of the ratings and to provide feedback to improve the effectiveness of test generation.

- **Refinement of Ratings**: In every round the ratings are updated based on the test added in that round. If the test happens to be a failing run, the statements whose suspiciousness and confidence values decrease from the previous round are not faulty in most probability. This is under the assumption that the new failing test also covers the same code faults as the initial run. Hence the ratings of these statements are not augmented (i.e. **Susp**$_{sattar}$**(s) = Susp**$_{Tar}$**(s)**). When the added test violates a subset of the constraints violated by the initial failing run, it probably covers a subset of the faults exposed by the initial run. In such a case, the above heuristic is not applied since it may reduce the suspiciousness of the faulty statements present in the initial failure

trace and not covered by the new test.

- **Feedback to customize test generation strategy**: Analysis of the ratings also helps improve the test input generation strategy on-the-fly. Statements heuristically determined to be non-faulty could be removed from the suspect list. The feedback of this updated suspect list to the test input generation module improves the effectiveness of the similarity criterion. We also asses the performance of the tests heuristically to identify redundant tests. Big fault-revealing inputs, such as long lists, lead to the generation of a series of failing runs with very similar code coverage, providing little benefit in refining the ratings. Hence the ratings are observed periodically to detect such series of redundant tests. Subsequently, the test input generation strategy switches to generating tests which differ from the previous tests not only in their total code coverage, but specifically in their coverage of the suspicious statements (`NEW_STRATEGY` in Listing4.2).

User control (Optional):

The incremental generation and processing of tests is continued until no more test cases with unique coverage can be generated. The user can optionally be presented with the output after a specified number of rounds and if he is satisfied with the precision he can choose to stop further refinement. This helps in cases when additional tests could reduce the localization accuracy, such as passing runs covering the faulty statement. Since we generate test cases that are as close to the initial failing run as possible, the first few tests could be assumed to cover the same

code faults in most probability (in multi-fault scenarios). But as the rounds increase, the chance of hitting new faults increases and user-control could aid in stopping the process before the generation of these tests.

4.3.3 Discussion of Correctness

In this section, we argue that our algorithm produces a suspect list that does not miss any faulty statement, assigns the highest suspiciousness value to the faulty statement(s), and in most cases ranks them higher than actually correct statements. We assume that the provided specification is not erroneous, the translation of code to logic is correct.

For single fault cases, the MUC is guaranteed to include the faulty statements responsible for the violation. The suspiciousness ratings of the statements in the suspect list are all incremented by 1.0. Hence the faulty statements would be assigned the highest suspiciousness value of 2.0. There may statements in the initial suspect list that are actually correct. Our test-generation strategy is geared to generate passing tests that aid in reducing the ratings of these statements.

We focus on localizing all faults exposed by the initial failure trace. When there are multiple faults occur simultaneously in a single failure trace, we require each of the faults to violate a distinct specification constraint. Our iterative processing of MUCs aids in producing a suspect list that contains all the faulty statements in the trace. Violated specifications analysis helps eliminate failing tests that cover faults not exposed by the initial trace. This prevents the assignment of lower ratings to the faulty statements due to interference from other faults.

4.4 Evaluation

We address the following research questions in the evaluation.

RQ1: Does SAT-TAR perform better than existing pure spectra-based localization and pure SAT-based analysis of specifications, for different types and number of faults and varying complexity of specifications?

RQ2: How much do the individual components of our localization algorithm, a) test generation using UNSAT core, and b) augmentation of ratings using UNSAT core, contribute to its effectiveness?

4.4.1 Candidates

Please refer Appendix B for code-snippets used for the evaluation.

BST.add(Integer k): Binary search tree (BST) is a data structure with complex structural properties, commonly used in applications requiring efficient search. The data structure invariants include uniqueness of each element, acyclicity with respect to `left`, `right` and `parent` pointers, size correctness, and search constraints. The `add` method inserts a node at an appropriate position in the tree based on the input value.

BaseTree.addChild(Tree t): **ANother Tool for Language Recognition (ANTLR)** [86] is a commonly used open source tool to build recognizers, interpreters, and compilers from grammars (DaCapo benchmarks [15]). It is an excellent candidate because it uses various kinds of data structures, specially trees, as its backbone. `BaseTree` class implements a generic tree structure customized to parse

Table 4.5: Single Fault Results. SAT: pure MUC based localization, Tar^{Rand}: Tarantula with a randomly generated suite, Tar^{DT}: Tarantula with tests generated by [8], and SAT-TAR (*user - SAT-TAR with user-control*).

Fault	*SAT*	*Tar^{Rand}*	*Tar^{DT}*	*SAT-TAR*	*SAT-TAR*
	#Tests=1 Rank	#Tests=10 Rank	<#Tests,Rank>		#Tests (user)
BST-sizeErr1	6	7.3	<6, 9.4>	<6, 2>	2
BST-sizeErr2	9	7.3	<6, 10>	<6, 5.4>	5
BST-parentErr	11	6.1	<6, 10.8>	<6, 2>	4
BST-rightErr	11	1.2	<6, 1>	<6, 1>	3
BST-traverseErr	7	2.4	<6, 4.2>	<6, 2>	3
BST-branchErr	11	6.1	<6, 6.3>	<6, 1>	4
ANT-parentErr1	3	2	<4, 3.4>	<4, 1>	3
ANT-parentErr2	4	2	<4, 3.9>	<4, 2.1>	3
ANT-childparErr1	8	6.3	<4, 6.6>	<4, 3.6>	2
ANT-childparErr2	9	6.3	<4, 6.4>	<4, 4.1>	3
ANT-childErr	7	6	<4, 6.8>	<4, 1.8>	2
ANT-loopErr	10	6	<5, 6>	<5, 4>	1

Table 4.6: Results of 2 restricted variants of SAT-TAR.

Fault	***Tar^{DT} + SAT***	SAT-TAR	***Tar^{MUC}***	*Tar^{DT}*
	Rank			
BST-sizeErr1	2.6	2	9	9.4
BST-sizeErr2	7	5.4	9	10
BST-parentErr	8.2	2	2	10.8
BST-rightErr	1	1	1	1
BST-traverseErr	4	2	2	4.2
BST-branchErr	3.2	1	2	6.3

input grammars. Each instance maintains its children as a list, each child is in turn a tree and has a pointer to its `parent`. Every tree node may also contain a `token` field which represents the payload. The `addChild(Tree t)` is the main method used to build all tree structures. Based on the comments and the program logic, we derived the following specifications: acyclicity of the children list, accurate parent-child relationships, and addition of child without any unwarranted modifications to the tree structure.

Table 4.7: Multiple faults localization.

Multi-Fault	Tar^{Rand}	*SAT-TAR*		*SAT* #Tests=1
	Rank <#Tests,(r_1,...,r_n)>	#Tests loc_1,loc_2	Rank r_1,...,r_n	Rank r_1,...,r_n
BSTrootparErr	<10,(16.3,7.2)>	2,3	2,3	5,10
BSTrootsizrghtErr	<10,(6.6,9.1,16.3)>	5,4	4,5,2	12,12,5
ANTtwoErrs	<10,(10.3,2.3)>	2,3	5,1	9,3
SLLsizremErr	<4,(6,1)>	4	5,1	9,9

4.4.2 Experiment Setup

Fault scenarios: We seeded different types of faults in both these methods (Table 4.5), including incorrect updates to data structure fields, local variables which impact the traversal of the tree, branch conditions and loop indices. Some faults lead to the violation of just one constraint such as `BST-parentErr`, which only violates the constraint `n = n. left`.parent`, while others such as `BST-rightErr` violate many constraints, involving almost all fields of the data structure. Error names ending with labels 1 and 2 represent the same code fault but with different post-condition specifications, impacting the UNSAT core. For instance `BST-sizeErr1` violates only `size` = size + 1`, while `BST-sizeErr2`

violates `size` =#this.root`.*(right` + left`)`, producing a bigger core and suspect list. Similarly, for `ANT-parentErr2`, the post-condition only checks if for every node the parent pointers have been set correctly for all its children, while `Err1` also checks if the input tree has been added correctly as a child. We also seeded more than one faults simultaneously to simulate multi-fault scenarios (Table 4.7).

Localization techniques: The spectra-based localization approaches

(**Tar**^{Rand},**Tar**^{DT}) use the same suspiciousness and confidence formulas for localization as Tarantula. For **Tar**^{Rand}, we used the state-of-the-art test input generation technique for data structures, Korat [16], to generate tests exhaustively up to a size such that there was a failing run for every fault. For each fault, we used a code version containing only that fault and selected 10 tests randomly (ensuring the inclusion of at least one failing run). For Tar^{DT}, the test-suite was generated using the directed test generation algorithm [8]. We implemented the algorithm using Forge, with SAT as the back-end technology, instead of concolic execution as done previously. This enables an efficient application of the technique on data structure programs with complex invariants. We employed a prototype implementation of our algorithm, built on top of the Forge framework, to obtain results for **SAT-TAR**. In both SAT-TAR and Tar^{DT}, tests are incrementally added to the suite. In order to compare the test-suite effectiveness, we used the same number of tests for Tar^{DT} as that produced by SAT-TAR to localize a particular fault. **SAT** is the pure UNSAT core analysis based technique which assigns the same suspiciousness value of 1.0 to all statements in a suspect list obtained by mapping from the MUC.

Metrics: The effectiveness of localization was measured by assigning a **rank** to the faulty statement (Section 2) based on its position in the descending order of the suspiciousness and confidence ratings of all the statements. The lower the rank of the faulty statement the better the localization.

Kodkod with *MiniSAT* as the backend SAT solver was employed to obtain the minimal unsatisfiable cores. All the experiments were run on a system with 2.50GHz Core 2 Duo processor and 4.00GB RAM running Windows 7. The results

are an average of 10 runs. All the localization approaches, except Tar^{Rand}, start with a single failing run. Hence for every run, we used bounded verification (Forge) to generate a fault-revealing input of random size.

4.4.3 Result Discussion

Table 4.5 shows the ranks assigned to the faulty statements for each fault under each technique. The # Tests for Tar^{DT} and SAT-TAR is the minimum number of tests amongst the 10 runs (for the run with the fault-revealing input of smallest size).

RQ1: **SAT-TAR vs Tar**: For all types of faults, our approach consistently produces lower ranks using lesser number of tests than Tar^{Rand}. For most of the faults, the best possible ranking is obtained using the first few tests and the algorithm could ideally be stopped at that point under user-control mode (user in Table 4.5). The precision is better than Tar^{DT} as well, despite the latter approach using a test-suite produced specifically for effective localization.

The use of UNSAT core helps filter out statements not related to the specification violation. For instance, in `BST-sizeErr1`, MUC analysis helps narrow down to 6 statements responsible for the violation, from a total of 14 statements. Even in faults not directly updating fields appearing in the specifications, such as `ANT-loopErr` and `BST-branchErr`, around 30% of statements are eliminated based on MUC analysis.

`BST-sizeErr2` is one of the many cases where our heuristic refinement of ratings proves beneficial. On processing two failing test cases, one adding a right

child to the tree and the other a left child, statements updating the `right` and `left` fields are rightfully eliminated from being faulty.

`ANT-parentErr1` is an instance, where our test-generation strategy produces a passing test most similar to the initial failing trace in the second round, resulting in the faulty statement being ranked first. The randomly selected tests (Tar^{Rand}) comprise of many failing tests with similar coverage which assign the same ratings to almost all statements.

Multiple faults: Table 4.7 shows the results for multi-fault scenarios ($r_1, ..., r_n$ represent ranks of faulty statements 1 to n, loc_1,loc_2 indicate the separate localizations runs for each failure trace). The test-suite used by Tar^{Rand} contains failing tests exposing different faults simultaneously which decreases the quality of localization due to interference. Our approach works on a single failure trace and builds a suitable suite to localize the faults exposed by that trace.

Consider `BSTrootparErr`, containing two faults simultaneously in `BST.add`; `BSTrootErr` and `BSTparentErr` (Figure B.2a). Two distinct failing tests cover each of these faults and violate two distinct constraints; `one this.root'` and `all n :n.left'.parent' = n` respectively. Our approach processes each of these failing tests separately and eliminates each of these tests from the other fault's localization. The faulty statements are thus assigned lower ranks in their respective localizations as compared to the ranks assigned to them by Tar^{Rand} using a test-suite containing both the failing tests.

SAT-TAR vs SAT: The high ranks of faulty statements (SAT in Tables 4.5 and 4.7),

highlight the poor precision of the localization based on pure SAT-based analysis of specifications. Specifically, `BST-sizeErr2`, `ANT-parentErr2`,

and `ANT-childparErr2` have exactly the same faults as their counterpart error cases ending with 1. However MUC analysis returns longer suspect lists for these cases either due to the increase in the complexity or decrease in the completeness of the post-condition specifications. On the other hand, SAT-TAR employs the coverage of additional tests to refine the ratings in the initial suspect list and hence the precision does not suffer much.

RQ2: To address RQ2, we compare SAT-TAR with two restricted variants, each including one specific component of the technique but not the other; i) **Tar**^{DT} **+ SAT**: This represents a mere aggregation of the ratings obtained from the stand-alone application of pure spectra-based localization and SAT-based analysis (Susp = **Susp**$^{DT}_{Tar}$ **+ Susp**$_{SAT}$), ii) **Tar**^{MUC}: This represents the use of MUC analysis only for test generation (Susp = **Susp**$^{MUC}_{Tar}$). SAT-TAR includes both the components, Susp = **Susp**$^{MUC}_{Tar}$ **+ Susp**$_{SAT}$.

Comparing the results of **Tar**^{DT} **+ SAT** with **SAT-TAR** (Table 4.6), shows that in most cases, our technique performs better than a technique that merely aggregates the ratings from pure spectra-based and SAT-based localizations. This highlights the benefits of our strategy to specifically generate tests using the information gained from MUC analysis. The results of SAT-TAR are better than **Tar**^{MUC}, indicating the benefit of UNSAT core based augmentation of suspiciousness ratings. Specifically, in cases such as `BST-sizeErr`, where the tests produced by our algorithm and the counterpart technique (*DT*) are almost the same, MUC analysis based

filtering helps lower the ranks significantly.

JTopas Case Study:

JTopas [1] is a Java library used for parsing arbitrary text data such as HTML, XML, programming language source code so on. We used SAT-TAR to localize a fairly complex fault (from version 0.4 of the application in SIR [2]) and another fault seeded by us. We tested the Tokenizer class by asserting that the parsed output satisfies the following two properties i) the '@' sign is followed by a keyword inside a Javadoc comment, ii) the number of open and closed braces are the same. Assuming that the test code was error free, we performed modular analysis of the `nextToken` method, which parses the input at the next position into an appropriate token. The code of this method with the called helper methods inlined, was encoded in the Forge Intermediate Language [34] with a total LOC of 100. **SAT-TAR** handled the size and complexity of the code quite well and localized the faults precisely (ranked the faulty statements amongst the top 2).

```
1 Input1:              Input2:
2 /**@author*/          /*{}*/
```

Listing 4.3: Fault-revealing inputs for the two faults in JTopas.

Performance: Table 4.8 shows the average times taken by Tar^{DT}, pure SAT-based analysis and SAT-TAR for localizing each of the errors. SAT-TAR consumes, on an average, about twice as much time as pure spectra-based localization. More than 97% of this time is spent in the extraction of the unsatisfiable core. Hence the difference in times is more pronounced for applications with large specifications such as ANTLR, wherein every node of the tree has multiple children. The large

Table 4.8: Localization Times.

Fault	*Tar*^{DT}	*SAT*	*SAT-TAR*
	Time in secs		
BST-sizeErr1	2.91	8.17	8.30
BST-sizeErr2	2.56	7.14	7.27
BST-parentErr	2.97	7.69	7.75
BST-rightErr	3.98	7.23	7.37
BST-traverseErr	2.27	7.41	7.51
BST-branchErr	4.57	7.40	7.54
ANT-parentErr1	3.98	15.42	15.73
ANT-parentErr2	5.31	15.61	15.96
ANT-childparErr1	19.16	95.42	95.88
ANT-childparErr2	18.13	24.27	24.55
ANT-childErr	20.12	36.4	40.10
ANT-loopErr	43.78	15.54	15.96
	Time in mins		
JTOPAS-Err1	15.77	1.43	18.12
JTOPAS-Err2	1.80	1.45	4.48

size of the source code impacts the performance of all three techniques for the JTOPAS application.

Threats to Validity: Use of our implementations of the **SAT** technique and the directed test generation algorithm (DT used in **Tar**^{DT}) may impact the construct and conclusion validity of our experiments. However, comparing the test-suite generated by SAT-TAR with the SAT-based implementation of DT, aids in rightly attributing the differences in the results to purely algorithmic differences, without being impacted by the back-end technology. The number of test cases used for techniques Tar^{Rand} and SAT differ from SAT-TAR. However, they represent typical applications of these existing approaches in practise. Representatives of the candidate programs may impact the external validity of the results.

4.5 Related Work

In this section we present a detailed discussion and comparison of our approach with other approaches that are closest to ours.

Directed Test Generation for Effective Fault Localization (DT): Recent work [8] presents a test-generation strategy specifically directed at localizing faults. This approach is similar to *SAT-TAR* in its attempt to specifically build a suite that contains tests that are *similar* to the initial failing test. However, we differ from this technique in the following manner, which aids in producing better localization.

Firstly, *DT* employs concolic execution [94] to keep track of path-constraints and generate test inputs covering new portions of code. The scalability of this approach has not been evaluated on data structure methods, wherein a `repOk` method would have to be symbolically executed before every method invocation to generate valid inputs. Our approach represents both code and specifications in relational logic which aids in the use of SAT for efficient generation of the test cases.

Secondly, the similarity score of a trace in the DT approach is based on the number of conditionals it covers in common with the failure trace that have the same truth value. On the other hand, our similarity metric compares the coverage of the statements deemed suspicious by MUC analysis. The benefit of our metric is highlighted in in the comparison **Tar**^{MUC} **vs Tar**^{DT} (Table 4.5). In `BST-branchErr`, a test case inserting a node into an empty list is the passing test case that is most effective in refining the precision of localization. The similarity score assigned to this test using our metric which is based on the coverage of the suspicious statements is higher than the score assigned to it using the counterpart technique, which is solely based on the truth values of the path conditions covered. Hence this test case is not included in the suite for Tar^{DT}.

Thirdly, the counterpart approach is used to generate a pre-determined set of

tests upfront before the localization phase begins. The advantage of the feedback-based improvement of test-generation strategy (employed in **SAT-TAR**) is highlighted in the localization of the JTopas faults. When the input file (Input1 in Listing 4.3) is parsed, the assertion on the first property (Section 4.4) fails, despite the fact that the token `author` following the '@' sign is a keyword. The first set of test inputs generated, comprised of different types of tokens (separator, space, etc.) substituted before and after '@author', leading to a series of similar failing runs. On detecting that even after processing 3 failing tests, the suspiciousness of the statements in the suspect list did not change, the test generation strategy switched to producing a run differing from the previous runs in its coverage of the suspicious statements. A passing test case was produced shooting up the precision of localization. The faulty statement in the `test4Normal` helper method which wrongly sets the type of the `author` token as `Normal` instead of `Keyword` is assigned a rank of 2. Tar^{DT}, on the other hand, ends up generating series of 5 similar failing tests shooting up the rank to 29.

Correctness Oracle based augmentation of Tarantula's ratings: Artzi et al. present an effective localization approach for web applications [9], that is related to ours in terms of augmenting the ratings of Tarantula based on an oracle for correctness. However, this technique is very specific to errors in PHP applications generating HTML pages. Also, only statements for which Tarantula assigns a rating greater than 0.5 are augmented, which is not very effective for typical errors in data structure methods wherein both passing and failing runs cover the erroneous statements. It is not explicitly shown if multiple error cases are handled effectively

by the technique. The paper presents a novel condition modeling approach to catch omitted branch statements. We envisage the use of MUC based analysis for the same purpose. For instance, if the update to size were omitted in the list example, the violated `size-ok` constraint would indicate that the size field was possibly wrongly updated or not updated at all. Analyzing the code statements that the MUC contains, would likely show the omission error.

Chapter 5

SAT based Program Repair

.

This chapter is based on our paper, "*Specification-based program repair using SAT*", published at the 17^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011) [45]. Dr. Malik and Dr. Khurshid were the co-authors on this paper. Dr. Malik helped me with the experiments and Dr. Khurshid provided useful insights to improve the approach and refined many sections of the paper.

5.1 Summary

Our key insight is to transform a faulty program into a *nondeterministic* program and use SAT to prune the nondeterminism in the ensuing program to transform it into a correct program with respect to the given specification. The key novelty of our work is the support for rich behavioral specifications, which precisely specify expected behavior, e.g., the `delete` method of a binary search tree only removes the given key from the input tree and does not introduce spurious keys into the tree, as well as preserves acyclicity of the tree and the other class invariants, and the use of these specifications in pruning the state space for efficient generation of program

statements.

We present a framework that embodies our approach and provides repair of Java programs using specifications written in the Alloy specification language. Given a fault revealing input, which yields an output structure that violates the post-condition and a candidate list of faulty statements, we parameterize each statement with variables that take a nondeterministic value from the domain of their respective types. A conjunction of the fixed pre-state, nondeterministic code, and post-condition is solved using SAT to prune the nondeterminism. The solution generated by SAT is abstracted to a list of program expressions, which are then iteratively filtered using bounded verification.

5.2 Illustrative Overview

Let us consider our running example of the faulty `delete` method of a singly linked list (Listing. 1.1). It uses two local pointers, `prev` and `lst`, to traverse the list and bypass the relevant node by setting the `next` of `prev` to the next of `lst`. Let us consider the post-condition specification (Figure 1.3), that checks that the specified value has been removed from the list in addition to ensuring that the invariants such as acyclicity of the list are maintained. In the faulty version, on statement 5 `prev.next` is assigned to `lst` instead of `lst.next`.

A scope bounded verification technique such as Forge can statically verify this method against the post-condition and produce a fault-revealing input that exposes the fault (Figure 1.2). The pre-state is a linked list with two nodes ($List_0$ is an instance of a list from the `LinkedList` domain, N_0 and N_1 are instances from the

`Node` domain). In the corresponding erroneous post-state, node N_1 with element value 2 is still present in the list since the next pointer of N_0 points to N_1 rather than $null$. Let us assume the presence of a fault localization technique which identifies the statement `prev.next = lst` as being faulty.

Our aim is to correct this assignment such that for the given input in the pre-state, there exists a path through the code that yields an output that satisfies the post-condition. To accomplish this, we replace the operands of the assignment operator with new variables which can take any value from the `Node` domain or can be $null$. The statement would get altered to $V_{lhs}.next = V_{rhs}$, where V_{lhs} and V_{rhs} are the newly introduced variables. We then use SAT to find a solution for the conjunction of the new formula corresponding to the altered code and the post-condition specification. The values of the relations in the pre-state are fixed to correspond to the fault revealing input as follows, $header =< List_0, N_0 >$, $next =< N_0, N_1 >$, $prev =< N_1, N_0 >$, $elem = < N_0, 1 >, < N_1, 2 >$. The solver searches the bounded state-space of the failing execution for suitable valuations to the newly introduced variables, such that an output state is produced that satisfies the post-condition for the given fixed pre-state. In our example, these would be $V_{lhs} = N_0$ and $V_{rhs} = null$.

These concrete state values are then abstracted to programming language expressions. For instance, in the example, the value of the local variables before the erroneous statement would be, $lst = N_1$, $prev = N_0$ and $this = List_0$. The expressions yielding the value for $V_{rhs} = null$ would be `prev.next.next`, `lst.next`, `this.header.next.next`. Similarly, a list of possible expressions can be arrived

at for $V_{lhs} = N_0$. Each of these expressions yields an altered program statement producing correct output for the specific fault revealing input. The altered program is then validated for all other inputs within a pre-defined scope using the bounded-verification technique,Forge. This process filters out candidates that may work for the specific input that revealed the fault but may not be generalizable enough to work correctly on other inputs. In our example, Forge would throw a counter-example on using the altered statement `prev.next = this.header.next.next` since it would not work for lists having more than two nodes. When no counter-example is thrown, it indicates that the altered statement works correctly on all inputs within the scope and is considered correct repair of the fault. In this example, `prev.next = prev.next.next` and `prev.next = lst.next` emerge as the correct statements satisfying all inputs.

5.3 Framework Details

Fig. 5.1 gives an overview of our framework for program repair. We assume the presence of a verification module indicating the presence of faults in the program. In our implementation, we have employed bounded verification (Forge) to generate the smallest fault revealing input (s_t) that satisfies the pre-condition of the method and yields an output (s_t') that violates the post-condition. A trace, comprising of the code statements in the path traversed for the given input, is also produced. We also assume the presence of a fault localization scheme, which yields a minimal list of possibly faulty statements. Please note that the technique works on the Control Flow Graph representation of the program (CFG) and the relational model

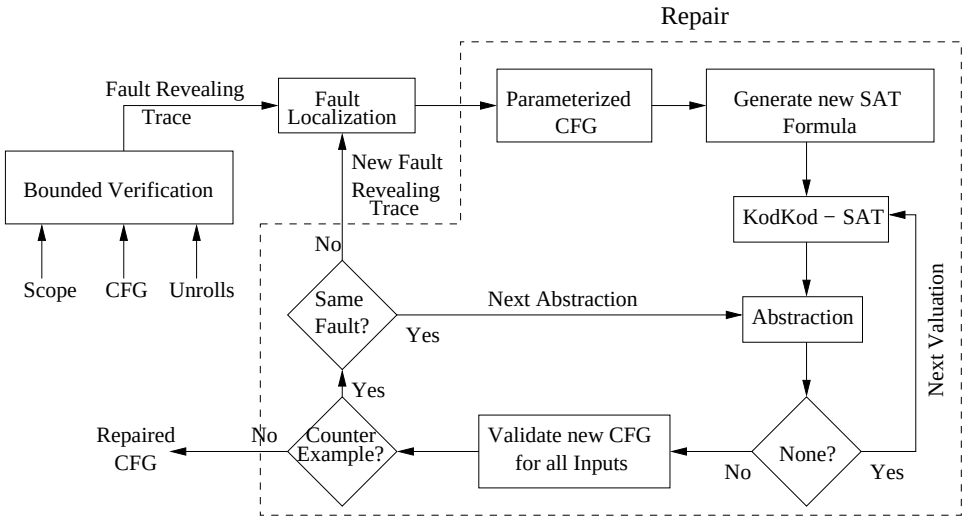

Figure 5.1: Overview of our repair framework.

view of the state.

Given a counter-example from the verification module and a list of suspicious statements $S_1, \ldots, S_m$, the first step performed by the repair module is to parameterize each of these statements. The operands in the statement are replaced with non-deterministic variables. For instance, consider an assignment statement of the form, $x.\{f_1.f_2.\cdots.f_{n-1}\}.f_n = y$. The presence of a commission error locally in this statement indicates that either the source variable x, one or more of the subsequently de-referenced fields $f_1, f_2, \ldots, f_{n-1}$ or the target variable y have been specified wrongly. The altered statement would be $V_{lhs}.f_n = V_{rhs}$, where V_{lhs} and V_{rhs} can be $null$ or can take any value from the domains A and B respectively, if f_n is a relation mapping type A to B. Similarly a branch statement such as `x`

> y, would be altered to $V_{lhs} > V_{rhs}$ and a variable update $y = x$ to $y = V_{rhs}$. A new set of variables would thus get defined for each statement in suspect list, $\{V_{lhs}^{S_i}, V_{rhs}^{S_i}\}, \forall i \in \{1, \ldots, m\}$.

New constraints for the code with the newly introduced variables are generated in the next step. The input is fixed to the fault revealing structure by binding the pre-state relations exactly to the values in s_t when supplying the formula to the Kodkod engine. A formula made up of the conjunction of the fixed pre-state, the formula for non-deterministic code and the post-condition, $pre\text{-}state \wedge code\text{-}constraints \wedge post\text{-}condition$, is fed to the SAT solver. The solver looks for suitable valuations to the new variables such that a valid trace through the code is produced for the specified input (s_t) that yields an output (s_t'') that satisfies the post-condition. If $V_{lhs}^{S_i}$ is a variable of type `Node`, then a possible solution could bind it to any concrete state value in its domain, $V_{lhs}^{S_i} = n, \forall n \in \{N_0, \ldots, N_{scope-1}\}$ (scope is the user-defined bound on the number of instances that the `Node` domain can contain). Hence every new variable in every statement in the suspect list would be bound to a concrete state value as follows, $V_{lhs}^{S_i} = C_{lhs}^{S_i}, V_{rhs}^{S_i} = C_{rhs}^{S_i}, \forall i \in \{1, \ldots, m\}$, where $C_{lhs}^{S_i}$ and $C_{rhs}^{S_i}$ are the state values at every statement.

The ensuing abstraction module attempts to map every concrete valuation to local variables or expressions that contain that value at that particular program point. The expressions are constructed by applying iterative field de-references to local variables. We ensure that we do not run into loops by hitting the same object repeatedly. Stating it formally, if for variable v, and fields $f_1, \ldots, f_k$, $v.f_1.\cdots.f_k$ evaluates to v, then for any expression e that evaluates to v, the generated expression

will not take the form $e.f_1.\cdots.f_k.e'$, rather it will take the form $v.e'$. Please note that when a parameterized statement occurs inside a loop, there would be several occurrences of it in the unrolled computation graph. Each occurrence could have different state valuations, however, their abstractions should yield the same expression. Hence we only consider the last occurrence to perform the abstraction. A new CFG is generated with the altered statements. In the event that no abstractions can be produced for a particular concrete valuation, we invoke SAT again to check if there are alternate correct valuations for the given input.

Please note that if SAT is unable to find suitable valuations that yield a satisfying solution for the given input or none of the valuations produced by SAT can be abstracted, it indicates that the fault cannot be corrected by altering the statements in the suspect list. The reason for this could either be that some statements had been omitted in the original code or that the initial list of suspicious statements is inaccurate. At this stage, manual inspection is required to determine the actual reason. Omission errors could be corrected by the tool using "templates" of the missing statements inserted by the user at appropriate locations. In case of the suspect list being incomplete, a feedback could be provided to the localization module to alter the set of possibly faulty statements.

In the validation phase, we use scope bounded checking to systematically ensure that all inputs covering the altered statements yield outputs satisfying the post-condition. If a counter-example is detected, the new fault revealing input and the corresponding trace, along with the altered CFG are fed back into the fault localization module. Based on the faults causing the new counter-example, an altered

or the same set of statements would get short-listed in the suspect list. The process is repeated until no counter-example is detected in the validation phase. There could be faults in the code not detected during the initial verification or during the above mentioned validation process. These may be present in a portion of code not covered by the traces of the previously detected faults. Hence as a final step, we verify that the CFG is correct for all inputs covering the entire code. Thus, the repair process iteratively corrects all faults in the code. In case, all the faulty statements can be guaranteed to be present in the suspect list, the entire CFG can be checked whenever an abstraction is generated. When a counter-example is detected, the subsequent abstractions for the same set of statements could be systematically checked until one that satisfies all inputs is produced.

5.3.1 Discussion of Correctness

In this section, we argue that the repair algorithm terminates, is accurate (yields statements that are correct for all inputs within a given scope or bound) and covers all types of commission errors. We start with a scenario with the most number of assumptions and go down progressively relaxing the constraints for subsequent scenarios. The basic assumptions are that the specifications are accurate and that the faults are not present in the operator or constant values specified in a statement. Omission errors can be identified but out of scope for automatic correction.

Scenario 1: Only commission errors, All code faults have been identified in the initial verification, and the fault localization scheme produces the exact list of faulty

statements.

We attempt to correct a statement by changing the values read or written by it. In a statement of the form $x.f_n = y$, we do not parameterize the field being updated (f_n). If f_n had been specified wrongly instead of f_n‘, then either the update to f_n‘ field should have been omitted or specified wrongly elsewhere. As indicated earlier, automatic correction of omission errors are beyond the scope of this work. In the latter case, the erroneous update to f_n‘ should also be in the list of faulty statements. The repair algorithm would correct the update to f_n‘ as well as make the erroneous update to f_n a no-op.

Our algorithm can correct destructive updates to class fields, faulty updates to local variables, and faulty branch conditions. We do not alter the operators or constant values specified in the statement. We hypothesize that such errors can typically be caught by the programmer during his initial unit testing phase itself, whereas detection of errors relating to the state depends on the input being used. This may get missed during manual review or testing and a bounded verification technique which systematically checks for all possible input structures is adept in detecting such errors. However, if by altering the placement of the operands, the semantics of a wrongly specified operator can be made equivalent to the correct operator, then we can correct such errors. For instance, if `x < y` has been wrongly specified as `x > y`, the corrected statement would be `y > x`.

The abstraction phase ensures that the algorithm finds values in the state-space that not just yield the correct output state, but are mappable to correct program expressions. For instance, in a branch statement there could be a number of

possible values for the operands that could yield the boolean outcome required to direct the control flow in the right direction. However, only a particular combination would be mappable to the variables at that program point. Following from our assumptions for this scenario that all the faulty statements are guaranteed to be in the initial suspect list, the program variant that would work correctly for all inputs should be in the list derived by analyzing the initial fault revealing input. Hence, the repair process is guaranteed to terminate. The guarantee on the completeness and correctness of the repaired program is the same as that provided by other specification based scope bounded verification techniques. The repaired statements are correct with respect to the post-condition specifications for all inputs within the highest scope used to validate the repaired program.

Scenario 2: Fault localization scheme produces a possibly faulty list. All the statements in the list (correct and faulty) are parameterized. This weakens the code constraints leading to a number of iterations of valuations and abstractions before a repaired program satisfying all inputs could be found. We assume that a good localization scheme would yield a minimal number of possibly faulty statements hence the process should terminate in a reasonable amount of time and the repaired program would not differ too much from the intended algorithm. In case, all the statements in the suspect list are actually not faulty, SAT would not be able to generate a valuation which yields a correct output for the given input. This case is identified and the user notified.

Scenario 3: There could be faults other than those revealed by the initial verification. New faults could get revealed when an abstraction for the first fault is being

validated. As explained earlier, the new input and the repaired CFG are fed back into the localization module. If the first fault had not been corrected, the program could fail again for the same fault in the subsequent validation phases. In order to prevent exploration of expressions that had already been considered and invalidated previously for a particular fault, we maintain a list of *already_seen* expressions which are avoided in the subsequent abstraction phases.

5.4 Evaluation

This section presents two case studies on (i) Binary Search Tree `insert` method and (ii) `addChild` method of the ANTLR application [86], described in Section 4.4.1. Please refer Appendix B for the code snippets used in the evaluation.

5.4.1 Metrics

The efficiency of the technique was measured by the total time taken to repair a program starting from the time a counter-example and suspect list of statements were fed into the repair module. This included the time to correct all faults (commission) in the code satisfying all inputs within the supplied scope. Since the time taken by the SAT solver to systematically search for correct valuations and validate repair suggestions is a major factor adding to the repair time, we also measured the number of calls to the SAT solver. The repaired statements were manually verified for accuracy. They were considered to be correct if they were semantically similar to the statements in the correct implementation of the respective algorithms.

Our repair technique is implemented on top of the Forge framework [33].

Table 5.1: Case Study Results: P1 - `BST.insert`, P2 - ANTLR `BaseTree.addChild`. Errors categorized into 3 scenarios described in section 5.4.2. The number of actually faulty and correct statements in the suspect list of fault localization(FL) scheme are enumerated. Description highlights the type of the faulty statements. Efficiency measured by Repair Time and number of SAT Calls. Accuracy measured by (i) whether a fix was obtained, (ii) was the repaired statement exactly same as in correct implementation. Every result is an average of 5 runs(rounded to nearest whole number).

Name	Scr#	Error#	FL Scheme Output		Type of Stmts	Repair Time(secs)	# SAT Calls	Accuracy
			# Faulty	# Correct				
P1	1	1	1	0	Assign Stmt	3	2	$\surd$, Same
		2a	1	0	Branch stmt	34	114	$\surd$, Diff
		2b	1	0	Branch stmt	4	2	$\surd$, Same
		3a	1	0	Assign stmt	5	2	$\surd$, Diff
		3b	1	0	Assign stmt	5	4	$\surd$, Same
		4a	1	0	Branch stmt	12	96	$\surd$, Diff
		4b	1	0	Branch stmt	4	2	$\surd$, Same
		4c	1	0	Loop condition	1	2	$\surd$, Same
		5	2	0	Branch, Assign stmts	7	5	$\surd$, Same
		6	2	0	Assign stmts	5	3	$\surd$, Same
	2	7	1	2	Branch, Assign stmts	15	21	$\surd$, Same
		8	2	1	Branch, Assign stmts	6	2	$\surd$, Same
		9	1	1	Assign stmts	11	2	$\surd$, Same
	3	10	4	0	Branch, Assign stmts	6	8	$\surd$, Same
		11	2	0	Branch, Assign stmts	26	9	$\surd$, Same
		12	2	1	Branch, Assign stmts	33	14	$\surd$, Same
		13	2	1	Assign, Branch stmts	14	24	$\surd$, Same
		14	0	2	Omission error	NA	NA	NA
P2	1	1	2	0	Assign Stmt	71	2	$\surd$, Diff
	2	2	2	2	Branch, Assign stmts	1	5	$\surd$, Same

The pre, post-conditions and CFG of the methods were encoded in the Forge Intermediate Representation (FIR). Scope bounded verification using Forge was used to automatically detect code faults. The suspect list of possibly faulty statements was generated manually. The input scope and number of loop unrolls were manually fixed to the maximum values required to produce repaired statements that would be correct for all inputs irrespective of the bounds. MINISAT was the SAT solver used. We ran the experiments on a system with 2.50GHz Core 2 Duo processor and 4.00GB RAM running Windows 7.

5.4.2 Results

Table 5.1 enlists the different types of errors seeded into the `BST insert` and `ANTLR addChild` methods. Figures B.1a and B.1b show the code snippets of the methods with the errors seeded. The errors have been categorized into 3 scenarios as described below.

Binary Search Tree insert:

Scenario 1: Only commission errors, All code faults identified in the initial verification, suspect list contains the exact list of faulty statements.

In the first eight errors only one statement is faulty. In errors 1 and 3, the fault lies in the operands of an assignment statement. Error 1 involves a wrong update to the `parent` field causing a cycle in the output tree. The faults in the latter case, assign wrong values to local variables `x` and `y` respectively. For instance, in error 3a, the variable `x` is assigned to `x.right` instead of `x.left` if the input value `k < x.key` inside the loop. This impacts the point at which the new node

gets added thus breaking the constraints on the `key` field. The repaired statement produced is `x = y.left` which is semantically similar to the expected expression, since the variable `y` is always assigned to `x` before this statement executes. A bigger scope and larger number of unrools are required to detect the the faults in error 3. This increases the search space for the solver to find correct valuations resulting in higher repair time.

Errors 2 and 4 involve faulty branch statements present outside and inside a loop respectively. In Errors 2a and 4a, the comparison operator is wrong and both the operands are parameterized as $V_{lhs} > V_{rhs}$. The search is on the integer domain and passes through many iterations before resulting in valuations that produce an expression satisfying all inputs. The final expression is semantically same as the correct implementation, with the operands interchanged (`y.key > k` vs `k < y.key`). In errors 2b and 4b, the expression specified in the branch condition is faulty. Hence only the variable in the expression is parameterized (`k <` V_{rhs}`.key`). The search on the `Node` domain results in correctly repaired statements with lesser number of SAT calls than a search on the bigger integer domain. In errors 5 and 6, combinations of branch and assignment statements are simultaneously faulty. Since the number of newly introduced variables are higher, the repair times are higher than the previous errors. Also, when there are more than one statements in fault, combinations of abstractions and valuations corresponding to each statement, need to be parsed to look for a solution satisfying all inputs.

Scenario 2: Fault localization scheme produces a possibly faulty list.

In this scenario, the suspect list also includes statements which are actually

correct. For instance, in error 7, the fault lies in an assignment statement which wrongly updates the `parent` of the inserted node similar to error 1, but the suspect list also includes 2 branch statements before this statement. In this case, all the operands of the 3 statements are parameterized leading to an increase in the search space and hence the repair time. It can be observed that as the percentage of actually faulty statements increases, the number of SAT calls decreases. In error 9, an assignment statement inside a loop wrongly updates a variable which is used by a subsequent branch statement. Owing to the data dependency between the statements, the number of possible combinations of correct valuations are less resulting in just 2 SAT calls, however, the large size of the fault revealing input increases the search time. The results were manually verified to ensure that the expressions assigned to the actually correct statements in the final repaired program were the same as before.

Scenario 3: There could be other faults than those revealed by the initial verification.

Not all code faults may get detected in the initial verification stage. For instance, in error 10, an input structure as small as an empty tree can expose the fault in the branch statement. However, when the correction for this fault is validated using a higher scope, wrong updates to the `parent` and the `right` fields get detected. The new fault-revealing input and the CFG corrected for the first fault are fed back into the fault localization scheme and the process repeats until these faults get corrected. When the entire program is checked again, a wrong update to the `left` field of the inserted node gets detected. The process repeats as explained

before until the last fault is corrected. This explains the 8 calls to SAT to correct this error scenario. Owing to the small size of the fault revealing inputs, the suspect list containing only the erroneous statements and the fact that in most runs, the first abstraction for every faulty statement happened to be the correct ones, the total repair took only 5 seconds. However, correction of subsequent errors necessitated more number of iterations leading to higher repair times, which got exacerbated when the suspect list also included statements not in error.

In the last case (error 14), the statement to update the `parent` field is omitted from the code, the localization scheme wrongly outputs the statements which update the `left` and `right` fields as being possibly faulty. The repair module is unable to find a valuation which produces a valid output for the fault revealing input. It thus displays a message stating that it could possibly be an omission error or the statements output by the localization scheme could be wrong.

ANTLR addChild:

The `addChild` method is much more complex than BST insert consisting of calls to 4 other methods, nested branch structures and total source code LOC of 45 (including called methods). The first fault consists of a faulty increment to the integer index of a loop. A local variable `j` is assigned the value of the integer index `i` plus 1 inside the loop. After every iteration, `i` is assigned `j + 1`, erroneously incrementing the index by 2 instead of 1. This fault requires an input of size 4 nodes and 2 unrolls of the loop to get detected. The error can be corrected by assigning `i` instead of `i + 1` to `j`, or `j` to `i` after every iteration, or `i + 1` to `i` (original implementation). We assumed that the suspect list includes both the assignment

statements. In the repaired version, `i` was assigned `i + 1` after every iteration and the assignment to `j` was not altered. The correction was performed in just 2 SAT calls but consumed 71 seconds due to the large state space.

The second scenario simulates a case wherein both a branch statement checking whether the current tree has any children or not and a statement updating the `parent` field of the added child tree are faulty. These faults are detected with a scope of 3 and unrolls 1. Two actually correct update statements are also included in the suspect list. Our technique is able to repair all the statements in one second.

5.5 Discussion

As can be observed from the results of the evaluation, our repair technique was successful in correcting faults in different types of statements and programming constructs. The technique was able to correct up to 4 code faults for `BST insert` within a worst case time of 33 seconds. The technique consumed a little more than a minute (worst case) to correct faults in `ANTLR addChild`, highlighting its ability to scale to code sizes in real world applications. Overall, we can infer that the repair time is more impacted by the size of the fault revealing input rather than the number of lines of source code. It has been empirically validated that faults in most data structure programs can be detected using small scopes [7]. This is an indication that our technique has the potential to be applicable for many real world programs using data structures.

One of the shortcomings of the technique is that the accuracy of the repaired statements is very closely tied to the correctness and completeness of the

post-condition constraints and the maximum value of scope used in the validation phase. For instance, in the second error scenario of `ANTLR addChild` (Figure B.1b), when the post-condition just checked the addition of a child tree, an arbitrary expression yielding `true` was substituted in place of the erroneous branch condition. Only when the specifications were strengthened to ensure that the child had been inserted at the correct position in the children list of the current tree, was an accurate branch condition obtained. Hence, the technique can be relied upon to produce near accurate repair suggestions, which need to be manually verified by the user. However, the user can either refine the constraints or the scope supplied to the tool to refine the accuracy of the corrections.

Following points are avenues for improvements in the repair algorithm; Erroneous constant values in a statement could be corrected by parameterizing them to find correct replacements but avoiding the abstraction step. Erroneous operators in a statement could be handled with the help of user-provided templates of possibly correct operators, similar to the handling of omission errors as described in Section 5.3. The number of iterations required to validate the abstractions for a particular valuation can be decreased by always starting with those involving direct use of local variables declared in the methods. Methods that manipulate input object graphs often use local variables as pointers into the input graphs for traversing them and accessing their desired components. Hence the probability of them being the correct abstractions would increase.

5.6 Related Work

We discuss two techniques most closely related to our work: program sketching using SAT [99] (Section 5.6.1) and program repair based on data structure repair [78] (Section 5.6.2).

5.6.1 Program sketching using SAT

Program synthesis using sketching [99] employs SAT solvers to generate parts of programs. The user gives partial programs that define the basic skeleton of the code. A SAT solver completes the implementation details by generating expressions to fill the ”holes” of the partial program. A basic difference between our work and program sketching is its intention. We are looking to perform repair or modifications to statements identified as being faulty while sketching aims to complete a given partial but fixed sketch of a program. An application of the sketching idea in the repair domain would be to use SAT to directly search in the space of program variants for one that would yield the correct output satisfying the specifications. Faults in data structure programs typically lead to an erroneous state that is in close neighborhood of the corrupt structure. Hence a local search in the state-space should yield a satisfying solution much faster than a technique that exhaustively enumerates all possible program variants. Moreover, the faults in recursive data structure implementations can be typically revealed using small sized inputs. Hence the scope of the possible state-space variants at a point would be lesser than the possible program variants.

To illustrate, consider our running example of the SLL `delete` method

(Figure 1.1). The sketching process starts with a random input structure (Figure 1.2). Assume that the user specified the details for the entire `delete` method except for the statement that updates `prev.next`. The user would then need to specify a generator stating all possible program expressions that can occur at that program point. Following the sketch language described in the synthesis paper, this would be specified using generators as shown below,

$$\#define\texttt{LHS} = \{\texttt{prev.((next)?.(next)?)}\}$$

$$\#define\texttt{RHS} = \{\texttt{(header|lst|prev).(next)?.(next)?|null}\}$$

```
LHS = RHS;
```

The list of possible expressions with the `LHS` fixed to `prev` would be, `prev = header, prev = header.next, prev = header.next.next, prev = lst, prev = lst.next, prev = lst.next.next, prev = prev, prev = prev.next, prev = prev.next.next`. Including the options of `prev.next` and `prev.next.next` in `LHS`, the number of possible expressions at the program point would double. Even for the small example with just 2 unknowns in 1 statement, the number of possible expressions are many. As the program size and complexity increases, the set of all possible program expressions that could be substituted at possibly faulty locations can become huge. In such cases, it would be faster to look for a correct structure in the concrete state space. The number of expressions mapping to the correct state value at a program point would be much lesser in number. As explained in Section 5.2, the number of options for the example would reduce to just three.

The synthesis through sketching technique employs an iterative process of validation and refinement (CEGIS) to filter out sketches that do not satisfy all inputs, similar to our validation phase. However since synthesis starts with a random input which may have poor coverage, it would require higher number of iterations to converge to a correct sketch. Since we start with the input that covers the faulty statement, the iterations should converge to a correct abstraction faster. Further, if the statements produced by fault localization can be guaranteed to be the only ones faulty, the correct expression must be present in the list of abstractions for the first input. Hence scanning this list would be faster than having to feedback every new counter-example into the system.

5.6.2 Program repair using data structure repair

On-the-fly repair of erroneous data structures is starting to gain more attention in the research community. Juzi [38] is a tool which performs symbolic execution of the class invariant to determine values for fields that would yield a structurally correct program state. A recent paper [78] presents a technique on how the repair actions on the state could be abstracted to program statements and thus aid in repair of the program. The main drawback of this technique is that it focuses on class invariants and does not handle the specification of a particular method. Hence the repaired program would yield an output that satisfies the invariants but may be fairly different from the intended behavior of the specific method. Further, in cases where the reachability of some nodes in the structure gets broken, Juzi may be unable to parse to the remaining nodes of the tree and hence fail to correct the

structure. Error 4c (Figure B.1a) in the `BST insert` method, highlights such a scenario wherein the loop condition used to parse into the tree structure being wrong, the new node gets wrongly added as the root of the tree. Our technique looks for a structure that satisfies the specific post-condition of method, which requires that the nodes in the pre-state also be present in the output structure. Hence a correct output tree structure gets produced.

Chapter 6

Program Repair using Machine Learning

This chapter describes projects where we have applied techniques from machine learning to repair database programs (Section 6.1) and imperative programs (Section 6.2).

6.1 Repair of database programs

This section describes our paper "*Data-Guided Repair of Selection Statements.*", published at the 36^{th} International Conference on Software Engineering (ICSE 2014 [43]. Dr. Saha, Dr. Chandra and Dr. Khurshid were the co-authors on this paper. Dr. Chandra mentored me in developing the approach and wrote many sections of the paper, while Dr. Saha helped me with the experiments and also formalized the algorithm. Dr. Khurshid provided useful insights and contributed to the writing of the paper.

6.1.1 Summary

A majority of enterprise software systems are database-centric programs. Defects in such programs, specifically in database manipulating statements, are expensive to fix and can require much human effort in understanding the interplay

```
1  SELECT CstId Price Year from OrderTab INTO
       itab
2
3  SORT itab by CstId
4  DEL from itab where Year <= 2009 and Price > 5

5  LOOP AT itab INTO wa
6    AT NEW CstId
7      amount=0
8    ENDAT
9    amount = amount + wa.Price
10   AT END CstId
11     WRITE wa.CstId amount
12   ENDAT
13 ENDLOOP
```

Figure 6.1: A sample ABAP code segment.

between traditional imperative code and database-centric logic. Automated tools to help diagnose these defects, and furthermore, to assist with fixing them can make a substantial reduction in the cost of developing and maintaining database-centric programs.

6.1.2 Illustrative Overview

Our specific focus is on SAP ERP systems, in which database-centric programming is carried out in a proprietary language called ABAP. ABAP contains SQL-like commands, but it mixes imperative code and SQL's declarative syntax. We introduce the essential constructs of ABAP that are relevant for this paper using a small example (Figure 6.1).

The meaning of this ABAP code segment is straightforward. At line 1, it reads all rows from a database called `OrderTab` into an internal table called `itab`. The `SORT` statement sorts this internal table by `CstId`, which is the key. The `DEL` statement at line 4 removes from `itab` those rows that match the condition described in the

statement. The `LOOP` at line 5 iterates over `itab`. When it encounters a new `CstId`— that is when `AT NEW` at line 6 is true— it resets an accumulator called `amount`, and it prints the accumulated amount when the last record of that `CstId` has been visited; this is done when `AT END` on line 10 is true. (`AT NEW` and `AT END` help with key-wise aggregation akin to the SQL GROUP-BY construct.)

Suppose the program in Figure 6.1 is run on the database in Table 6.1. The rows marked '+' are retained in `itab` after the `DEL` statement. The output of the program is, which unfortunately differs from the expect output, also shown:

ID	Amount	Expected Amount
1	51	51
2	25	**32**
3	7	7

The bug arises from an error in the condition of the `DEL` statement, which causes the first row for `CstId` 2 to be incorrectly deleted (shown by a bold '**-**').

We can think of the `DEL` statement as a (equivalent) SELECT statement: `SELECT * FROM itab WHERE Year > 2009 OR Price <= 5`. We call such a defect a *selection* bug, because the bug is due to an incorrect WHERE condition in a SELECT statement. The problem is to find an alternate WHERE condition for the faulty SELECT statement (whose location is assumed to be known), so that the entire output, corresponding to each of the keys, is correct.

Table 6.1: A sample input to program in Figure 6.1. The last column with '+'/'-' is not a part of the input table.

CstId	Price	Year	
1	20	2012	+
1	16	2011	+
1	12	2001	-
1	10	2002	-
1	15	2011	+
2	7	2005	**-**
2	13	2007	-
2	15	2010	+
2	10	2011	+
3	4	2012	+
3	3	2009	+
3	9	2001	-

Continued on right . . .

Selection bugs are common in ABAP programs. In fact, many database statements in ABAP programs allow a selection condition, and therefore, are vulnerable to a selection bug. For example, the ABAP `READ` and `DELETE ADJACENT` statements can be modeled as selection statements. Based on our experience working with practitioners in IBM Global Business Services, about 25% of the ABAP code level defects have to do with selection. Such bugs typically do not reveal themselves while testing with limited set of data that is available in the test environment. The production environment has a lot more data and therefore exposes the corner cases that do not show up while testing. Moreover, the lack of an automated test-case generation tool for this framework is another reason why such bugs are not discovered while testing. Therefore, techniques that can help in fixing defective selection statements are of much value.

Note that in our setting of debugging ABAP programs, the process starts with the *end user* of this software filing a bug report, citing a deviation of the actual output from the expected output on given input data. Thus, the expected output of the program is already known to the programmer (or the maintainer). As we shall see, the challenge here is in *determining the correct behavior of the defective* SELECT *statement from the expected output of the entire program*, and in *determining an alternate* WHERE *condition for the selection that would match the correct behavior*.

An obvious technique to generate a correct selection condition would be to explore the space of syntactic mutations of the buggy condition. Because of the possible presence of data values in the clauses that constitute the conditions, the

search space for a mutation-based technique is immense. The size of the mutation search space for our suite of benchmarks is reported in Section 6.1.4. This makes the technique very inefficient for real use. In comparison, our approach, presented next, sidesteps the drawbacks of a mutation-based approach.

A Data-Driven Approach: An Overview

Our key observation is that in real-world data, there is information latent in the distribution of data that can be useful to repair the WHERE condition efficiently. Syntactic search completely ignores this latent information. In this work, we show that it is possible to find a good repair suggestion efficiently if we took advantage of this information.

In our approach, we first discover the correct behavior of the selection statement on the failure causing input data, and then find an alternate selection statement that exhibits the correct behavior. Our approach leverages the distribution of input data in both of these phases.

A defective selection statement assigns incorrect + or - labels to some of the rows of the input; for example, some of the rows for `CstId` 2 do not have the correct labels. To discover the correct behavior of the defective selection statement, we need to search through all possible assignments of labels to rows that have possible incorrect labels. Our technique carries out this search efficiently by taking advantage of the distribution of data.

Since part of the output is correct, we can assume that the rows that contributed to that part are labeled correctly; the remaining rows are possibly misla-

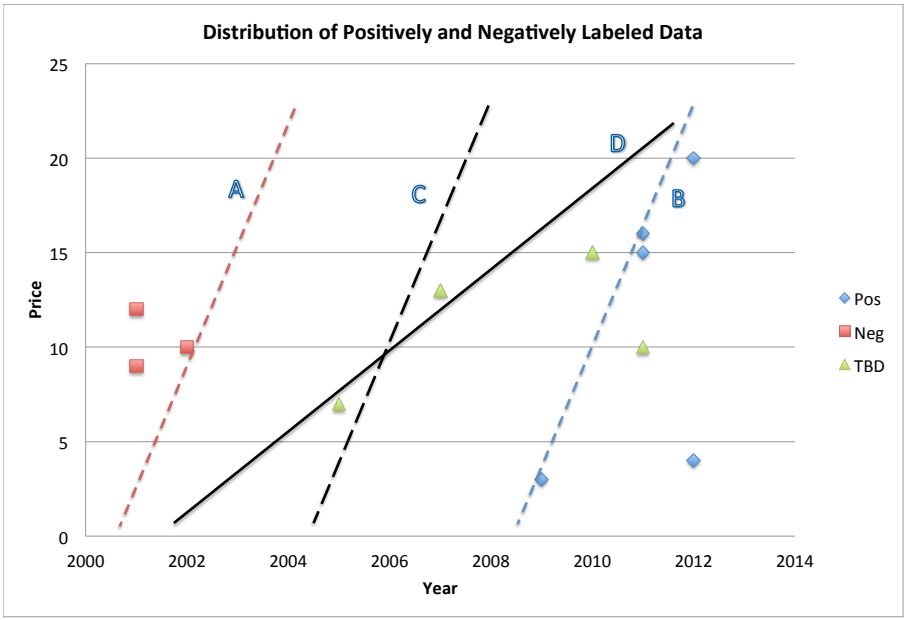

Figure 6.2: Distribution of data in Table 6.1

beled. Our premise is that a possibly mislabeled row that is geometrically close to a correctly-labeled row is likely to require the same label. Obviously, this notion of proximity is not guaranteed to produce the correct labels, but they can serve as a very good starting point from which to carry out the search for the right labeling. This is exactly what we do: use a labeling computed on the basis of geometric proximity, but fix it up based on local search around that labeling.

In Figure 6.2, data of passing keys in Table 6.1 is shown with diamonds for positively labeled data and squares for negatively labeled data. For rows belonging to `CstId` 2, whose labels as generated by the WHERE condition are suspect, data is shown with a triangle (unlabeled). Assuming that points that are spatially close are

likely to be labeled similarly, an assignment of a positive label to the two unlabeled points on the right can be done with relatively high confidence. The two unlabeled data points in the middle of the chart could go either way, so an assignment of a negative label to the two points in the middle can be done only with low confidence. Table 6.2 shows a sample assignment of predicted labels to the failing rows. We generated these predications using an implementation of support vector machines (SVM [13, 103]). Informally, SVM creates separating lines A, B, and their center–C–as its best effort on how to separate positively and negatively labeled data. In this particular example, the line D would have be the perfect separator; so, the two points that are close to separator C are predicted incorrectly.

The incorrect lower-confidence predictions can be fixed up by a combinatorial search for labels, until we obtain correct labels for all the rows (correctness validated by the final output matching the expected output). We carry out this search iteratively, starting with the row with the least confident prediction. In realistic problem sizes, this strategy, which takes advantage of data distribution, is significantly more efficient than combinatorial search on all the rows.

Once we have the correct labels for all the rows, the problem reduces to that of finding a function (a *classifier*) that attaches correct + or - labels to rows depending on the contents of the row. As mentioned before, it is difficult to find such

Table 6.2: Predicted label assignment for the failing rows

CstId	Price	Year	Predicted Label	Correct Label	Confidence
2	7	2005	-	+	0.3
2	13	2007	+	-	0.2
2	15	2010	+	+	0.9
2	10	2011	+	+	0.9

a function by looking for syntactic variations on the existing WHERE condition. In general, a vast number of different functions could produce the correct labels for a given set of rows. Not all of these would be close to the one intended by the programmer, because they may be *overfitted* to the data, in the sense that those functions may not label as-yet-unseen data correctly. A common heuristic is to look for a *compact* function, because it is more likely to be generalizable, and therefore (presumably) correct.

We use the distribution of data to guide the search for a compact function using a well-known technique known as decision-tree learning. This technique performs a greedy search over a space of functions, being guided by the distribution of data and the label for each row of data. The way this greedy search works is to first identify a predicate that classifies *most* of the data correctly, and then iteratively identify additional predicates as required to classify the residual data.

In the running example, first it would realize that partitioning the rows of the table on the basis of $Year \leq 2008$ gives the maximum, though not perfect, efficacy in terms of clubbing the + and - labeled rows in *distinct* partitions. See Figure 6.3(a), which shows the result of splitting on the basis of $Year \leq 2008$; again, squares are negatively labeled points and diamonds are positively labeled points, and the data is for the correct labels on all rows of Table 6.1. An alternate split, say on the basis of $Price \leq 10$, shown in Fig 6.3(b) is less effective in clubbing the + and - labeled data in distinct partitions. The ID3 decision-tree learning algorithm [82] captures this intuition using the concept of information entropy and automatically chooses the most advantageous splitting predicate.

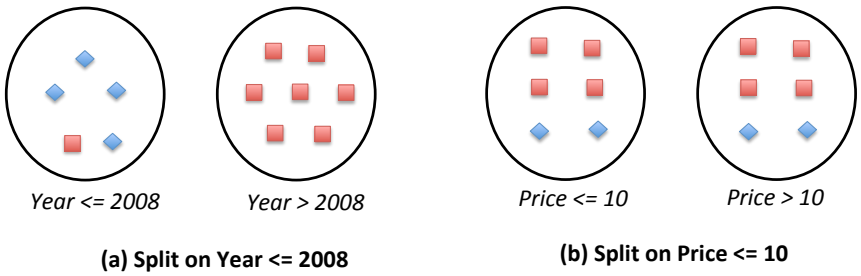

Figure 6.3: Splitting the data points on the basis of alternative conditions

In the partition for which $Year \leq 2008$, the maximum efficacy is obtained by further partitioning on the basis of $Price \leq 8$, at which point, all positives are perfectly separated from the negatives. In the partition for which $Year > 2008$, all rows are positive regardless of the price. This decision tree is (written as conjunctions of clauses on paths from root to +ve leaf nodes, and disjunction over such paths): $Year > 2008 \vee (Year \leq 2008 \wedge Price \leq 8)$. By DeMorgan's laws, this simplifies to the following condition: $Year > 2008 \vee Price \leq 8$ Comparing this to the previous incorrect WHERE condition, we see that while the learned WHERE clause for `Year` is slightly but gratuitously different, for `Price` it is crucially different.

Our technique manages to find "natural" conditions that a programmer would have written, and therefore ends up offering useful repair suggestions. We attribute this desirable property to our data-guided approach. A WHERE condition that a programmer writes is intended to classify *regions* of data uniformly, as opposed to cherry picking points in the data space and classifying them individually by some

```
1    Repair (s, Out, CorrectOut, key_fields,
         Trace)
2
3    //initialize
4    find FKeys, PKeys, sFKeys, sPKeys
5
6    //step 1: predict labels for failing key
         rows
7
8    projPrediction, inputPrediction, Map,
         prodTbl =
9        predict(sFKeys, sPKeys, sIn, sOut, s)
10
11   //step 2: Correct label computation
12   sCorrectOutSet = label(s, Trace, sFKeys,
13                     projPrediction)
14
15   //step 3: WHERE condition generation
16   generate_conditions(sCorrectOutSet,
17       prodTbl,Map,inputPrediction)
```

Figure 6.4: Algorithm: Generating repair suggestions for selection statement

complex conditional logic. This is the reason that predictions based on spatial proximity work well, and also the reason that the heuristic of finding a compact decision tree works.

6.1.3 Algorithm Details

In this section we describe the repair algorithm in detail. As shown in Figure 6.4, the algorithm has three major steps: 1) exploit the distribution of data to predict the selection result for the input rows of the faulty selection statement, 2) verify the predictions and if required determine the selection result using combinatorial search for the parts where the predictions are incorrect, and 3) generate correct conditions using decision-tree learning algorithm.

We illustrate the steps of the algorithm using an example shown in Figure 6.5. The example is a slight variation of the example presented in Section 6.1.2

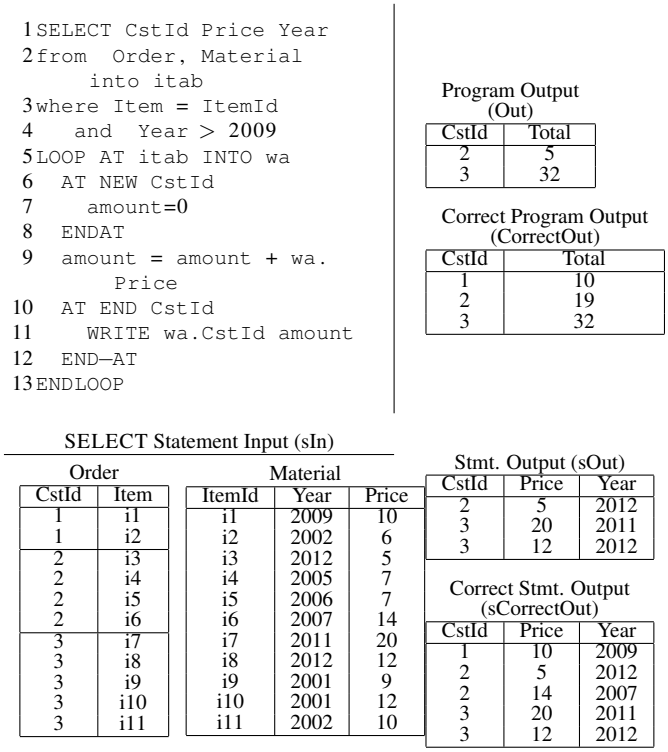

Figure 6.5: An ABAP program and data; s is the SELECT statement

to illustrate some salient features of the algorithm. The SELECT statement (shown in Lines 1-4) is the faulty statement. Below are the entities used in the algorithm:

- `Trace`: the execution trace which produces incorrect output.
- `s`: the trace occurrence of the faulty statement
- `sIn`, `sOut`: the set of input tables and current output of `s`
- `Out`: incorrect program output
- `CorrectOut`: expected correct program output
- `sCorrectOut`: a correct output of `s` which can produce `CorrectOut`

```
1  predict (sFKeys,sPKeys,sIn,sOut,s)
2    svm_in = empty
3    prodTbl = empty  [s.t. empty X t=t]
4    for each t ∈ sIn, prodTbl = prodTbl X t
5
6    //creating classification for prediction
7    for each r ∈ prodTbl
8        r_class = 0 if r.key ∈ sFKeys
9        r_class = +1 if r.key ∈ sPKeys,
             selection(r,s)
10       r_class = −1 if r.key ∈ sPKeys, !
             selection(r,s)
11       svm_in.add(<r,r_class>)
12
13   //label input rows
14   Set<r,r_predict> in_prediction = SVM(svm_
         in)
15       where r_predict is a real number
16
17   //label projected rows (for Joined Table
         in Input)
18   for each r ∈ prodTbl st. r.key ∈ sFKeys
19       projected_row = projection(r,s)
20       Map(projected_row).add(r)
21   for each projectedRow p in Map.keySet
22       predict(p)=Max(in_prediction(r))|r ∈
             Map(p)
23   return predict, in_prediction, Map,
         prodTbl
```

Figure 6.6: Algorithm: Prediction

- `key_fields`: a set of fields which uniquely identifies each row of `Out`

The `Out`, `CorrectOut`, `sIn`, `sOut`, `sCorrectOut` for the example are shown in Figure 6.5.

In the domain of data-centric programs, each row in the output is identified by a set of field-value pairs, called *key*. The algorithm compares the current program output (`Out`) and the expected output (`CorrectOut`) to determine a set of failing keys (`FKeys`) and passing keys (`PKeys`) for the program output. A passing key is a set of key_field-value pairs which identifies identical rows in `Out` and `CorrectOut`. A failing key is a set of key_field-value pairs which identifies a row which exists in `Out` but not

in `CorrectOut` or vice versa or identifies a row in `Out` and a row in `CorrectOut` which differ in at-least one non-key_field value. In the example, `CstId=3` is the passing key, whereas `CstId=1` and `CstId=2` are the failing keys. Note that, `CstId=1` corresponds to a missing row in the output.

All the input rows in `s` (obtained using the cartesian product of all tables in `sIn`), that directly or indirectly affect the failing rows (incorrect/missing/unwanted) in the program output are considered as *failing input rows* and the rest are considered as *passing input rows*. Such classification is performed based on a key-based dependency analysis between the passing and failing keys in the output of the program and the input rows in `s`. The set of passing and failing keys for `s` are denoted as `sPKeys` and `sFKeys`. In the example, rows corresponding to $\langle$`CstId,1`$\rangle$ and $\langle$`CstId,2`$\rangle$ are `sFKey` rows, and those corresponding to $\langle$`CstId,3`$\rangle$ are `sPKey` rows. As described in Section 6.1.2, the prediction algorithm predicts the selection result for the rows corresponding to `sFKeys` using the selection result for the `sPKey` rows.

Prediction of correct output of `s` The Function `predict` in Figure 6.6 first creates the input (`prodTbl`) by performing the Cartesian product of all input tables (in the example, `Material` and `Order`), then assigns label 0 (signifying rows whose values are to be predicted) to the input rows corresponding to `sFKeys`. The existing selection clause behaves (either selects or deselects) correctly for the `sPKey` rows. Each passing row in the input is thus classified as +1 if the row has been selected, and -1 if the row has not been selected, denoting known and correct classification (Lines 7-11). This forms the input to an SVM, which assigns to each input row with label 0, a

```
1  label(s, Trace, sFKeys, prediction)
2    model = code2model(Trace,s)
3    sCorrectOutSet = empty
4    for each sFKey ∈ sFKeys
5        threshold = 0
6        while(true)
7          label = empty
8          for each p ∈ prediction.KeySet
9              s.t. p.key=sFKey
10             predValue = prediction.get(p)
11             if |predValue|> threshold
12                label(p)=true if predValue >
                       0
13                label(p)=false if predValue
                       < 0
14         keyCorrectOutSet = SAT(label ∧
               Model
15         ∧ CorrectOut)
16         if keyCorrectOutSet is Empty
17             threshold = threshold +
18                                 ParamThresholdIncrement

19             if(threshold > maximum_
                   confidence_value)
20                 return failure
21             else
22                 continue
23         else
24             break
25       sCorrectOutSet = sCorrectOutSet X
             keyCorrectOutSet
26
27   Add passing key data to each selection in
        sCorrectOutSet
28   return  sCorrectOutSet
```

Figure 6.7: Algorithm: Labeling for failing keys

signed value called *prediction* (Line 14). The sign (called *label*) indicates whether the input row has been predicted to be selected (positive value) or not selected (negative value). The unsigned prediction value denotes the *confidence* associated with the prediction. In the example, the `prodTbl` contains 121 rows, 22 of them are for `CstId=1` and 44 rows are for `CstId=2`, which are marked 0. The remaining 55 rows for `CstId=3` are labeled +1 (22 rows) or -1 (33 rows).

Note that a SELECT statement first performs a *selection* of the rows in

`prodTbl`, followed by a *projection* to generate the output of `s` (*projected rows*). The set of input rows which correspond to a projected row, is called the *block* of the projected row. A projected row is in the output of s, if at-least one input row from its block is selected. SVM is employed to predict the outcome of the selection of the input rows. Our algorithm uses this to determine the prediction for the projected rows (Lines 18-22). The likelihood of a projected row being present in the output of s, is determined by the maximum likelihood of selection of the input rows in its block. The algorithm determines the prediction of the projected row as the maximum prediction (signed value) of all the rows in its block.

In the example, there are 121 rows in the `prodTbl`, which project to 33 rows, 11 rows per key. The predictions for the projected rows of `CstId=1` and `CstId=2`, are shown above.

CstId=1, projected rows

CstId	Price	Year	Pred.
1	10	2009	+0.8
1	6	2002	-0.4
1	5	2012	-1
1	7	2005	-1
1	7	2006	-1
1	14	2007	-1
1	20	2011	-1
1	12	2012	-1
1	9	2001	-1
1	12	2001	-1
1	10	2002	-1

CstId=2, projected rows

CstId	Price	Year	Pred.
2	10	2009	-1
2	6	2002	-1
2	5	2012	+1
2	7	2005	-0.1
2	7	2006	+0.1
2	14	2007	+0.2
2	20	2011	-1
2	12	2012	-1
2	9	2001	-1
2	12	2001	-1
2	10	2002	-1

The salient features of this step are summarized here:

- Determining passing and failing input rows and exploiting the data-distribution to predict selection results of the failing input rows.
- Mapping prediction values from input rows to the projected rows based on the maximum likelihood of selection.

Determining Correct Output of `s` The next step is to use these signed values to determine the part of `sCorrectOut` corresponding to the failing keys. The algorithm, described in Figure 6.7, tries to determine this per failing key (Line 4), for a reason described later. First, the output of `s` is created strictly as per the predicted labeling. The label is determined by the sign of prediction, which if positive denotes that the row is to be selected to the output, and not selected (label = false) if negative. If this does not yield the `CorrectOut`, then the projected rows corresponding to the `sFkeys` are gradually unlabeled. The decision whether a row would be unlabeled is done based the confidence of its prediction. A parameter `threshold` is used to gradually un-label low confidence projected rows. A projected row having confidence value above this `threshold` is labeled. Note that, the algorithm starts with `threshold` value zero to make all the projected rows labeled. In the running example, for `CstId=1`, the prediction selects only one row with `Price=10` (corresponding to item `i1`), whereas for `CstId=2`, it selects 3 rows with `Price` values 5 (`i3`), 7 (`i5`), and 14 (`i6`).

Next, we discuss how we verify whether the labeling based on predictions yields `sCorrectOut` and if not, how we use combinatorial search to label the rows whose predictions are relaxed based on the threshold adjustment.

If `s` contains a selection bug, then there exists a subset of its projected rows that can produce the expected program output. There can be combinatorial number of ways to create subsets of the projected rows. We employ a SAT solver to efficiently search for subset/s that lead to the final program output matching `CorrectOut`.

The scalability of the search depends on the input table size, which is usually large. To reduce space, we perform this search separately for every failing key.

Further, as described earlier, we rely on the predicted labelings for rows whose confidence is greater than `threshold` and perform combinatorial search only on the remaining unlabeled rows.

The algorithm creates a model of the code which can execute after `s` as a first order formula using Alloy (Line 2). The details of the translation is not provided in this paper, but is available at [44]. A SAT solver (Line 15) is used to validate if

```
1  generate_conditions(sCorrectOutSet,prodTbl,
       Map,inputPred)
2    ConditionSet = empty
3    for each sCorrectOut  ∈ sCorrectOutSet
4        weight(sCorrectOut) = average
             prediction value
5        of sFKey rows in answer
6    rankedSet=sort sCorrectOutSet based on
         weights
7    for each sCorrectOut ∈ rankedSet in
         order
8        c = getCondition(prodTbl,sCorrectOut,
9        Map,prediction)
10       ConditionSet.add(c)
11       if no more results required
12           return ConditionSet;
13
14 getCondition(prodTbl,sCorrectOut,Map,
       predictedlabels)
15   for each projectedRow p ∈ Map.keySet
16       if p ∈ sCorrectOut
17          max_pred = max.prediction of rows
                in Map.get(p)
18          for each r ∈ Map.get(p)
19            r_pred = predictedlabels.get(r)
20            class(r)=+1,
21                    if r_pred = max_pred
22                    or |r_pred|>threshold,
23                    r_pred > 0
24                    =−1,
25                    if |r_pred|>threshold,
26                    r_pred < 0
27       else
28           for each r ∈ Map.get(p)
29              class(r) = −1
30   return ID3(class)
```

Figure 6.8: Algorithm: Selection condition generation

the predicted labeling yields the correct output for the respective `sFkey`, and if not, perform combinatorial search to find such a labeling. It either returns no solution or returns a set of correct selections per failing key.

The SAT solver may not yield any satisfying truth assignments to the un-labeled projected rows for a failing key. This can be attributed to incorrect labels for rows marked based on the predictions. In this case, the algorithm increases the threshold to un-label some more low confident projected rows. Iteration based on adjusting threshold increases the domain size for combinatorial search. However, in our experiments we have seen that many iterations are seldom needed.

In the running example, SAT solver validates the prediction for `CstId=1`, but fails for `CstId=2` (yields total 26, expected 19). The low confidence projected rows for `CstId=2`, corresponding to

`ItemId=i4,i5,i6` are unlabeled. Combinatorial search in the second iteration yields two satisfactory combinations for `CstId=2` - `(i3, i4, i5)`, and `(i3, i6)`. Both of them yield total `Price` value 19 as in `CorrectOut`.

Combining all solutions for failing keys (Line 26) and adding the passing key selection (Line 27) the algorithm generates the following two `sCorrectOut`s.

CstId	Price	Year
1	10	2009
2	5	2012
2	14	2007
3	20	2011
3	12	2012

CstId	Price	Year
1	10	2009
2	5	2012
2	7	2005
2	7	2006
3	20	2011
3	12	2012

Note that, considering each failing key separately has another advantage apart from reducing the search space. It avoids unnecessary un-labeling of rows corresponding to keys for which the current labeling yields the correct output for

the respective keys.

The salient features of this step are summarized below:

- An iterative algorithm to label fewer projected rows based on predictions, if the current labeling does not yield `sCorrectOut`.
- Performing SAT-based combinatorial search on a per failing key basis, to determine the set of correct outputs for `s`.

Selection condition generation The algorithm for selection condition generation is presented in Figure 6.8. The generation of selection condition is performed using a home-grown implementation of the ID3 decision-tree learning algorithm which learns a classifier that provides 100% accurate classification for the training data. The Function `ID3` takes the selection result for the input rows to derive a compact condition satisfying the selection. The algorithm first ranks the set of correct outputs of s obtained in the previous step and calls the function `getCondition` corresponding to each `sCorrectOut` in order of the ranking. (Lines 7-12).

The Function `getCondition` classifies each input row of s with +1 or -1 based on the correct labeling for the projected rows. This is a reverse label mapping of what is done in Function `predict`. If the projected row is not present in the `sCorrectOut`, then all the input rows in its corresponding block should not be selected (Line 28). If a projected row is in output then one or more corresponding input rows can be selected. We mark the input row with the maximum prediction value as the one to be selected. (Line 21). For each of the other rows in the block, we use the

predicted label if its confidence is higher than the running `threshold` (Lines 23-26). Lesser confidence rows are not fed to the decision-tree learner. Finally ID3 is called with the classified input.

The conditions generated using this method are shown below:

- `(Item = ItemId) and (Year > 2006)`
- `(Item = ItemId) and (Year > 2008 or Price = 7)`

Note that the first condition is more preferable over the second as it is more compact and not overfitted to a specific value, hence it has more chances of being valid for unseen inputs. In general there can be many solutions given by combinatorial search. Since generating all possible repair suggestions is time consuming, our approach selects as input to the decision-tree learner the solution which has the potential to generate a good selection condition. The algorithm first ranks the solutions (Figure 6.8, Line 5) based on the average prediction value of the projected rows present in the respective `sCorrectOut`s. In our example, for `CstId=2`, the weight of the first solution, consisting of items `i3`, `i6` is (1+.2)/2=0.6 whereas it is (+1+0.1-0.1)/3=0.33 for the second solution corresponding to the selection `i3`, `i4`, `i5`. Based on our heuristics, the first solution is preferred over the second as the first solution has (`i4`) which is predicted to not be selected.

The basis of labeling input rows and ranking solutions is the same - it is possible to generate better quality condition by following the SVM prediction. There could be multiple outputs to the `SELECT` statement that yield the expected output of the program, however the `where` condition generated based on the labelings predicted by SVM is the most natural condition or closest to the ideal manual repair.

Section 6.1.4 contains experimental validation of this observation.

The salient features of the final step are summarized below:

- Ranking multiple solutions returned by combinatorial search.
- Labeling input rows based on the correct labels for the projected rows.
- Using ID3 to generate compact selection conditions.

More subtle variations of the algorithm, such as handling of multiple occurrences of the faulty statement and variations in threshold adjustment, are available in [44].

Pragmatics The application of SVMs to the problem at hand requires several steps of data conditioning. The main issue is that SVMs prefer to view data as numerical values for the purpose of distance computation. Relational database tables seldom contain data in this form. We discuss the problems and our solutions.

Nominal Attributes The table could contain *nominal* attributes, which are compared for equality, but not for order. For example, *State* (2-letter state abbreviation) is a nominal attribute. For such data, we introduce fresh columns, one for each distinct value of the nominal attribute that appears in tables. For *State*, we might introduce boolean attributes such as *State=AK* to *State=WY* and hide the original *State* attribute. At other times, data that looks like non-numeric data might need to be treated numerically. For example, dates have to be mapped to a numeric interval.

Key Attributes Keys are usually nominal data in that value-based proximity of two keys is not meaningful. Table joins created by Cartesian cross product of two tables

contain distinct key attributes coming from each of the tables in the join. Since it would require too many additional attributes to "de-nominalize" these key attributes, we instead include an additional boolean attribute that denotes the equality of these two keys (as it is common to have key equality comparison in SELECT statements for a natural join.)

Scaling It is typical in the use of SVMs to scale data to a normalized [0.0,1.0] range for each attribute. In case the range of data for a certain attribute is very large due to a few outliers, care is needed to prevent lower values being scaled down to too close to zero.

Selecting Relevant Attributes for ID3. The input tables of the buggy query typically have large number of attributes, many of which are irrelevant. We heuristically perform the following selection of potentially relevant attributes: (1) all attributes projected by the query (2) attributes that have been frequently used whenever this table has been used earlier in the code(such as key attributes). (3) attributes having the same data-type and overlapping values with the state variables at that execution point.

Seeding Synthetic Attributes in ID3. Decision-tree based algorithms are only equipped to learn clauses that compare attribute values with constants. However, WHERE conditions can contain comparison of two attributes. We seed binary-valued equality predicates between attributes as extra attributes into the learning algorithm. These predicates are seeded based on domain knowledge, for instance, attributes of the same data-type and having same range of values may be compared to select records.

Table 6.3: Summary of results. Times in Minutes (TO - 15mins)

Subjects	*# Passing input rows*	*# Failing input rows*	*Prediction Accuracy(%)*	*Correct label computation*				*Cond learning*	
				time	*search space*	*#itrs*	*#soln.*	*time*	*useful?*
Ex1	40129	10032	99.9 (10029)	4	6	3	1	2	Yes
Ex2	16641	30	36.6 (11)	2	$2^2 + 2^{17}$	2	32	10	Yes
Ex3	316	12	100 (12)	1	0	1	1	0.16	Yes
Ex4	274	58	25.8 (15)	3	2^{58}	30	1	0.03	No
Ex5	993	84	92.8 (78)	8	2^4	3	1	0.08	Yes
Ex6	90346	1816	99.8 (1814)	5	4	3	1	5	Yes
Ex7	13911	2	0 (0)	2	4	2	1	25	Yes

In the current state of our tool, each of the three modules of the algorithm are automated except as follows; `predict` requires data-conditioning as described above, `label` works based on a manual translation of the ABAP code-fragment to Forge Intermediate Language following the pseudo-code in [44], and `generate_conditions` requires heuristic selection and seeding of attributes.

6.1.4 Evaluation

This section first presents a summary of the experimental results which is subsequently explained using case studies of a select set of subject programs. Finally, it discusses relevant research questions and limitations of our approach.

We selected seven subject programs, which are fragments of ABAP code from industrial applications running on real data sets. The criteria employed to select these subjects were: (1) cover different types of selection bugs commonly found in bug reports (2) highlight the different characteristics and design decisions of the repair algorithm. Our evaluation covers different types of statements such as `SELECT` statements on single tables and `JOIN`s of tables with buggy `WHERE` clauses, and `DELETE` statements with buggy `COMPARING` clauses. The repair is applied on different types of

faults such as missing field checks, incorrect data value used in the conditions, incorrect/missing join conditions and also incorrect/missing range checks (equivalent to two missing conditions). The subjects fit into different scenarios (Section 6.1.4.1) highlighting the different features of our approach. The excerpts of real programs, the buggy selection conditions, and their fixes are available at [44]. The bugs in these programs are actual bugs that occurred in the past. In all cases we know the manual fix to the bug.

Our implementation uses the Alloy 4.2 [58] tool-set (specifically, Forge, Kodkod and miniSAT), SVM Light [62] in transductive learning mode, and a home-grown implementation of the ID3 decision-tree learning algorithm. All experiments were conducted in a 2.53Ghz CPU, 4GB RAM laptop running Ubuntu Linux.

The summary of our experimental results is shown in Table 6.3. For every candidate, we recorded the number of rows in the input table corresponding to the passing and failing keys, shown as *#Passing input rows* and *#Failing input rows* respectively in Table 6.3. Column 4 highlights the prediction accuracy, i.e. the % of failing rows for which the labels were predicted correctly. We also tabulate the total time for correct label computation (dominated by SAT solving times), the combinatorial search space, the total number of threshold relaxation iterations, and the number of `sCorrectOut`s generated. We also present the time to learn the WHERE condition. Finally, we state whether the repair suggestion was useful or not by comparing how close it was with the manual fix for the bug.

6.1.4.1 Example Scenarios

We describe details of applying our approach to four subjects to highlight some of its key characteristics and how it handles different types of faults.

Scenario 1: Repair without predictions. For simplicity, we start with *Ex3*, which illustrates a case in which just the basic approach for repair (without the use of predictions) can successfully produce a valid repair suggestion.

Consider the following buggy `select` statement in *Ex3*:

```
select * from ekbe into table tab_ekbe
           where ( vgabe eq '2' or vgabe eq '3' )
 //and ebeln in ebeln_range  Needed in the correct query
           order by ebeln ebelp.
```

The `where` clause is essentially missing two predicates in the form of a missing range-check predicate for the field `ebeln`. This error results in 12 unexpected rows in the output of the program.

Correct Label Computation. The incorrect rows in the program output correspond to 12 failing rows in the input `ekbe` table. SAT quickly finds that none of these records should get selected.

Although it is not required for this scenario, we did run SVM for predicting the labels for the 12 failing rows. The predictions were 100% accurate, and all the 12 records were assigned negative prediction values. We ran SAT marking them as dneg and the labeling was deemed satisfiable within a minute, i.e. not much savings in time over the basic approach.

Decision-tree learning. The condition learned was as given below and was correct

in not selecting exactly the 12 failing rows.

The generation of a comparison on `ebeln` conveys to the programmer that a bound check is missing. Indeed, the source code defines the constant `ebeln_range` as [`ebeln_low`, `ebeln_high`], but the buggy query fails to use it.

```
vgabe = '2'and
ebeln <= 4500000229
```

The generated repair could not infer the lower bound on `ebeln`, nor it could generate an additional condition on `vgabe`, because such conditions were not warranted by this specific input data. Nonetheless, the repair suggestion is useful in helping the programmer fix the problem. Our technique is intended to generate a useful repair suggestion for the programmer, as opposed to a perfect replacement.

Scenario 2. SELECT with table joins. This scenario illustrates a case where highly accurate prediction helps to significantly reduce the combinatorial search space for labeling.

The program (*Ex1*) creates a sales order report by calculating order amount and unbilled amount for each sales order. It first creates a table called `p_i_vbrp` using the following query:

```
select vbeln posnr aubel aupos matnr netwr
 from   vbrp, p_i_vbap
 into   table p_i_vbrp
 where  aubel = p_i_vbap—vbeln
 and    aupos = p_i_vbap—posnr
 // and netwr ¿ 0.  Needed in the correct query
```

However, the missing `netwr > 0` predicate from the WHERE condition causes incorrect `p_i_vbrp` formation, shown below for the key `aubel=102`.

```
Computed:                Expected:
aubel aupos  netwr       aubel aupos  netwr
102   20      0.00       102   20   8000.00
102   20   8000.00       102   30  11200.00
102   30      0.00
102   30  11200.00
```

The program logic after the SELECT statement reads the rows `<102, 20, 0>` and `<102, 30, 0>` corresponding to two `posnr` values `20` and `30`, instead of `<102, 20, 8000>` and `<102, 30, 11200>`, which it would have read from the correct version of the table. This leads to the incorrect output for the failing key `102`. Altogether there are 18 failing keys in this example.

Correct Label Computation.

There were 40129 passing input rows that were labeled as per their outcome in the existing execution. 10032 failing input rows were unlabeled. SVM attached a positive prediction to 19 unlabeled rows and a negative prediction to the remaining. As noted in Table 6.3, in this case the prediction is highly accurate (99.9%), leaving only 3 input rows incorrectly predicted. Each of these rows corresponds to a distinct failing key. and happens to be the one with the maximum prediction value in the block of the respective projected row. Thus for each of the 3 failing keys, the projected row was incorrectly labeled. For each key, the iterative threshold adjustment process was invoked until a correct solution was found. For example, for key `146`, initially both rows were marked negative (as shown below) leading to unsatisfiability.

```
aubel aupos netwr pred
146    10      0  -0.9
146    10     30  -0.3
```

With a threshold of 0.4, the second row with confidence less than 0.4 was assigned an unknown label (to be determined by SAT), while the first row was given nega-

tive label. The same was done with the other two failing key records. SAT assigned positive labels to these rows leading to satisfiable solutions. In all, this process completed in 4 minutes. Utilizing predictions produces a total search space of 3.2^1 = 6 for the SAT solver.

We also give an estimate of the combinatorial search space if predictions had not been used at all. For each of the 18 failing keys, on average there are 3 projected rows, where each projected row maps to a block size of 227 input rows. For each failing key, the state space for choosing the set of projected rows that yield the correct output is 2^3. Each solution comprising of projected rows, needs to be mapped to input rows of the joined table, before being fed to the ID3. The state space for this would be 227^3 in worst case, when the solution contains all three projected rows. Thus the total search space, in worst case, to generate correct condition without using any predictions would $18(2^3 + 227^3)$. As explained earlier, use of predictions helps reduce this space to just 6.

Decision-tree Learning. Decision-tree learning discovered the correct WHERE condition:

```
aubel = p_i_vbap-vbeln and
aupos = p_i_vbap-posnr
and netwr > 6
```

Note that our approach was able to learn the join condition. For each projected row, the input row that is selected from the corresponding block, is critical in determining the correct join condition. We would like to highlight that for the given data, the only input row in every block that satisfied the join condition, was the one with maximum prediction value in that block. Hence the selection of any other row from the block would have not lead to the discovery of the join condition. This

adds evidence to the fact that our design decision of selecting the row with maximum prediction value, would result in producing high quality conditions close to the ideal.

The constant discovered is 6 rather than 0, due to the distribution of the data. But it is a good repair suggestion since it points out an important missing clause.

Scenario 3. Use of predictions to rank candidate solutions. This scenario highlights that our repair algorithm is not restricted only to `select` statements, and further illustrates a case where predictions aid in reducing the space of candidate solutions on which decision-tree learning has to be performed.

The buggy statement in this example (*Ex2*) is a `DELETE` statement, shown below.

```
DELETE ADJACENT DUPLICATES FROM db_tab
   COMPARING kunnr matnr
   //arktx Needed in the correct query
```

The `DELETE ADJACENT DUPLICATES` statement deletes a row from the table that has same values in its immediately previous row for the fields specified in `COMPARING` clause. This could be modeled as an equivalent `select` statement as shown below.

```
select * from db_tab_rc as db_tab1, db_tab_rc as db_tab2
where db_tab1.rc = db_tab2.rc+1 and
db_tab1.kunnr = db_tab2.kunnr and
db_tab1.matnr = db_tab2.matnr and
// db_tab1.arktx = db_tab2.arktx Needed in correct query
```

Where `db_tab_rc` has an extra column `rc` in addition to all the columns of `db_tab`. It contains the same records as `db_tab` with the `rc` column populated with

the row number. This statement selects rows that would need to be *deleted* by the original statement. The code after the `DELETE` statement, in a nutshell, aggregates the `netwr` amounts corresponding to every unique value in `monat` field of `db_tab`. The output report had incorrect amounts displayed for two `monat` values - Sep2008 and Oct2008 (2 failing keys).

Correct Label Computation. The `db_tab` had 10 records with `monat` as Sep2008 and 20 records with Oct2008. Note that although the `select` is over a join and every row of `db_tab_rc` maps to a block of rows in the joined table, we know upfront the exact record that needs to be considered from every block. The only record that can be selected in the block corresponding to every failing row of `db_tab` is the one where `db_tab1.rc = db_tab2.rc+1` is satisfied (as this predicate should be present in the correct version of the query). Hence the input state space for SAT remains 10 and 20 respectively for the two failing keys and it becomes feasible to apply the basic approach of label computation without predictions.

SAT is invoked separately on the records for the 2 failing keys to determine the possible subsets of the records that would sum up to the respective expected final amounts. There are 8 possible subsets for Sep2008 and 32 possible subsets for Oct2008, leading to 256 possible correct labelings. It would be inefficient to generate 256 possible `where` clauses. This is where predictions aid in heuristically selecting the solution that is most likely to yield the ideal `where` clause. Note that this displays a scenario wherein label predictions aid in reducing the state-space of decision-tree generation even if the number of failing rows may be small enough for SAT to process.

Label Predictions-based solution ranking. There are only 80 positively labeled passing key records compare to 12691 negatively labeled records in the joined table. Hence the prediction accuracy in terms of the classification is low. However, the confidence of the incorrectly predicted labels were lower than the correctly predicted ones. We used predication-value based ranking of the solutions to select the desired solution for both for the two failing keys (comprising of 2 records and 17 records respectively).

Decision-tree Learning. The `where` clause learnt for the `select` statement was,

```
db_tab1.rc = db_tab2.rc + 1 and
db_tab1.kunnr = db_tab2.kunnr
and
db_tab1.arktx = db_tab2.arktx
```

As can be observed, the condition on `arktx` that was missing in the incorrect version is correctly discovered. However, the condition on `matnr` is missing from the learned clause. This is because the `matnr` values are equal for all adjacent records in which the other two conditions are also satisfied. This makes the learned `where` clause correct for the given input set.

Scenario 4. Impact of incorrect selection on passing keys. *Ex4* displays an interesting scenario which violates our assumption about the correctness of erroneous SELECT statement for the passing keys. The final output corresponding to passing keys is still correct but the SELECT acts incorrectly on some of them.

The erroneous SELECT statement given below leads to the inclusion of 58 extra records for the failing keys in the actual output of the program, compared to

the expected correct output.

```
select  ebeln  ebelp  belnr buzei  bewtp
        budat  matnr  werks ernam
 from ekbe into table it_ekbe
 where budat in s_crdate
  // AND vgabe = 1 Needed in the correct query
```

In this example, for the passing keys too, the erroneous SELECT statement selected some extra rows, but subsequently they got deleted by a DELETE statement in the program. Consequently, these passing keys yielded the correct final output anyway.

Correct Label Computation. The incorrect labeling for 16 passing key records where `vgabe = 1` impacts the accuracy of predictions as seen from Table 6.3. Hence the approach of using predictions to label records performs poorly. The algorithm passes through 4 iterations of threshold adjustment and produces correct solution only when all records are labeled based on SAT-based search.

Decision-tree Learning. The incorrect labeling of the passing key records impacts the WHERE clause condition learned. The condition is quite different from the one in the correct version of the code. It leads to the expected final output on this data set, but this is not a useful repair suggestion.

6.1.4.2 Discussion

Based on our experimental evaluation we address the following key research questions.

RQ1. Do the predictions based on the data distribution aid in finding the cor-

rect output for the failing keys efficiently?

In all cases, prediction based labeling of rows helps in determining a correct output state to the faulty statement within 8 minutes in the worst case.

The efficiency of our algorithm is attributed to high prediction accuracy which effectively reduces the combinatorial search space, and further design decisions such as: 1) an iterative threshold relaxation strategy which judiciously unlabels incorrect predictions. The low number of iterations (in most cases) suggest that there were only few incorrect predictions that needed to be labeled by SAT, 2) ranking of solutions based on predictions which saves the effort of generating conditions corresponding to all the solutions.

As noted in Scenario 2 (*Ex1*), in the absence of predictions, in the worst case, a combinatorial search based strategy has to explore huge search space to arrive at useful solution. Even for subjects that do not involve joins predictions-based labeling brings about significant reduction in search spaces: 87% reduction from a total of $2^{10} + 2^{20}$ for *Ex2*, and almost 100% reduction from a total of 2^{84} for *Ex5*.

RQ2. How useful are the repair suggestions?

The usefulness of the generated repair suggestions is summarized in the last column of Table 6.3. Except for *Ex4*, the repair suggestions were close to manual (ideal) fixes for the bugs. The reason for high quality of our repair suggestions can be attributed to the labeling of failing-key data based on its proximity to passing-key data which generates the conditions that classify regions of data uniformly, which is typical of WHERE clauses. Even though theoretically it is possible to generate

many `sCorrectOut`s which generate the expected correct output of the program, our approach only generates few of them based on the prediction. This in turn generates few good conditions which are close to the ideal fix.

For *Ex6*, we show below the manual fix and the condition generated by our approach using a solution that respects the predicted labels. In this case `p_p_werks` is a parameter to the program which had a value `GBS1`. As can be seen, this condition is very close the ideal fix.

Ideal fix for Ex6:	Condition learned from a solution based on predictions:
`vbeln = p_i_vbak-vbeln and werks = p_p_werks`	`vbeln = p_i_vbak-vbeln and werks = 'GBS1'`

We also show few conditions generated from other solutions that yield `CorrectOut` but do not use the predicted labels. Clearly such solutions are far away from the ideal fix.

Conditions learned from other solutions

`vbeln = p_i_vbak-vbeln and (werks = 'GBS1' and (vbeln <= 102.0 and (waer = 'EUR') or (waer = 'USD' and (posnr <= 15.0))) or (vbeln > 102.0))`	`vbeln = p_i_vbak-vbeln and (werks = 'GBS1' and (vbeln <= 102.0 and (netwr <= 3524.4) or (netwr > 6336.4)) or (vbeln > 102.0))`

RQ3. Is syntactic mutation technique feasible for real data?

To check the feasibility of mutation-based repair, we consider a repair strategy which checks if the WHERE condition could be corrected by either adding one clause, removing one existing clause, or replacing an existing clause with a new one. Clauses of the form `Field Operator Field`, `Field Operator Constant` and `Field Operator Variable` are considered as mutants.

Subjects	*Mutation*	
	Repair	
	time	*space*
Ex1	24.5	17663
Ex2	22	15860
Ex3	8.9	6434
Ex4	0.9	63
Ex5	56.1	40434
Ex6	25.4	18346
Ex7	200	147350

In almost all the cases the search space of the number of mutants is very huge (in the order of 40,000) leading to a blow-up in the worst-case exploration time (in the order of 40 hours assuming an average of 5 seconds to execute 1 mutant). The main reason being the large number of distinct values that could be compared in the clauses of the form `Field Operator Constant`. An algorithm that does not consider clauses that involve constants would work much faster, however it would be unsuccessful in discovering the correct WHERE condition for bugs such as in *Ex1*, *Ex3*, and *Ex7*.

Limitations 1. Our technique assumes that the incorrect selection criteria works correctly for keys that satisfy the final output correctness criteria. Violation of this assumption (*Ex4*) impacts the quality of the predictions and the WHERE condition learned.

2. Sufficient amount of diverse passing data is required to make the learning effective. For example, in *Ex7*, the failing rows were all erroneously predicted to be negatively labeled. This is because majority of the rows corresponding to the passing keys were negatively labeled; very few rows were positively labeled.

3. Attention must be paid to data conditioning, which currently uses heuristics based tuning (described in Pragmatics in Section 6.1.3) to arrive at good label prediction.

4. ID3 algorithm is designed to correctly label all training data. However, if the data

in the current execution is not representative enough, then the WHERE condition created may be overfitted to the data (*Ex4*). Techniques to avoid overfitting [82] compromise the accurate labeling of training data. Finding the right balance for our application is the subject of future work.

5. Our algorithm strives to generate the most compact classifier for the given data. In some cases, this could exclude clauses that would be in general necessary, but do not impact the outcome for the given data. To reiterate, our technique generates useful repair suggestions and not necessarily plug-and-play repairs.

6.2 Repair of imperative programs

This section describes our paper "*Use of Learning to perform Intricate Corrections to Faulty Branches.*", submitted to 30^{th} International conference on Automated Software Engineering (ASE 2015).

6.2.1 Summary

We present a novel approach to correct bugs that impact the branching behavior of imperative programs. Our insight is that mining the data-spectra (i.e., distribution of data in the state space of passing and failing tests) can aid in quick generation of a repaired branch condition that is heuristically close to an oracular fix. We define an integrated approach to repair Java programs based on (1) semi-supervised learning to predict expected behavior of failing tests based on similar passing tests, (2) state-space exploration to rectify incorrect predictions, and (3) decision tree learning to generate a classifier that represents the repaired condition.

Our case studies demonstrate the ability of the approach to generate intricate repairs accurately (including synthesis of omitted if statements and correcting loop constructs).

6.2.2 Illustrative Overview

Debugging code is often a tedious and error-prone process. In recent years, a number of research projects have introduced novel techniques to automate the two key tasks in debugging: (1) identifying the faulty lines of code, termed *fault localization* [87], [64], [48]; and modifying the faulty code to fix the faults, termed *program repair*, which is the focus of our work. A number of program repair techniques, e.g., those based on evolutionary algorithms [109], applying mutations to suspicious statements [18], performing program code transformations [29], have been demonstrated to hold potential. However, performing intricate repairs involving the addition or omission of multiple lines of code or correcting incorrect language constructs is still a challenge. Moreover, the large space of possible program variants impedes the efficiency and scalability of mutation or search based repair. Furthermore, the accuracy of the repair often is not generalizable beyond the given test-suite [85] or requires the presence of precise formal behavioral specifications [46].

Our focus in this paper is to provide efficient and accurate repair for the class of programs where the failure occurs due to wrong behavior of a branch. We employ learning techniques to generate accurate repairs even on structurally complex data, with the only oracle of correctness being the status of the tests. Conceptually, our

approach has two steps: (1) determine the behavior/s for the branch such that final outputs are correct; (2) synthesize a condition that represents a fix that generalizes beyond the given tests.

```
  public static boolean repOK(List list) {
1       List.Node l = list.header;
2       int cnt1 = 0;
3       boolean res = true;
4        while (cnt1 < list.size) {
           //ListERR1
5          if (l == l.next){
6               res=false;
7               break;}
8           l = l.next;
9           cnt1++;}
10      if (res == false)
11          return res;
       //size check
12      List.Node l1 = list.header;
13      int size=0;
14      while (l1 != null){
15          size++;
16          l1 = l1.next;}
17      if (size != list.size)
18          res=false;
19      return res;}
```

Listing 6.1: List.repOK

Problem Context:

Consider the repOK method for a doubly linked list data structure (Listing 1). A doubly linked list (DLL) is a popular data-structure that is made up of nodes connected to each other by a $next$ pointer. The $header$ of the list points to the first node and the $size$ field keeps count of the total number of nodes in the list. Each node has a $prev$ pointer that points to the node preceding. The repOK is a commonly used data-structure class method, which checks that the underlying structure conforms to its structural integrity constraints. The code in Listing 1 checks if

the list conforms to the *acyclicity* constraint (i.e. no node has its $next$ field pointing back into the list) and that the $size$ field correctly holds the number of nodes reachable from the $header$. It returns true if both these constraints are satisfied and false otherwise. There is a bug in the code on statement 5 which detects only self-loops ($l = l.next$) as cycles, however, loops involving two or more nodes, such as $l = l.next.next$, go undetected.

Test-suite 1 (Figure 6.9 is executed on this buggy code. It contains 9 tests, each of which invokes the repOK on a doubly linked list input (some valid and some invalid) and checks if the repOK returns true on valid inputs and false on the invalid ones. Owing to the bug in the code, 3 of the tests fail which return true on invalid inputs (having cycles other than self-loops).

Let us assume that the programmer is able to determine that the fault lies on statement 5. Despite this knowledge, it is challenging to determine what the correct condition should be. The functionality of detecting a loop in a DLL is considerably complex and the correction is non-trivial which cannot be generated using simple mutations. The space of possible program variants is large. For instance, the number of possible de-references of the $next$ or $prev$ fields on any variable could be as big as the number of nodes on the heap. Further, the only oracle of correctness being the final program output, the program needs to be executed to validate each possible variant. This makes applying search-based repair approaches expensive. The method also does not mutate the underlying structure and has a boolean return value. Hence, the output does not give any indication as to why the failure occurred.

Mining the data-spectra:

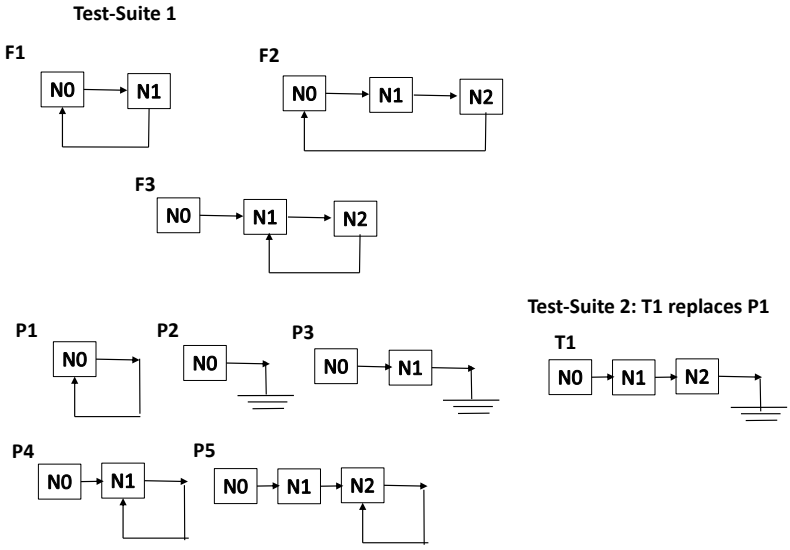

Figure 6.9: Test-Suites.

Our approach targets failures due to faulty branch behavior. The key idea is that there is information latent in the data-spectra of tests (distribution of data in the state space of passing and failing executions), which could be utilized to guide the generation of the correct branch condition.

The first step is to determine the correct behavior of the branch such that the failing tests pass. This has specific significance in cases where the faulty branch is invoked more than once in a single execution. For instance, in the above example, the faulty branch lies within a loop and hence there are multiple occurrences of the branch in a single failing test execution. One or more of these occurrences could have displayed faulty behavior leading to the wrong final output. There are multiple paths through the branch that need to be explored to discover the path/s leading to

Table 6.4: SVM input

Test	l	SVM input								
		cnt1	l $\neq$ null	l.next .next $\neq$ null	l.next $\neq$ null	l.prev .prev $\neq$ null	l.prev .next $\neq$ null	l.next $\neq$ null	l.prev $\neq$ null	l=l.next
F3	N0	0	1	1	1	0	0	1	0	0
	N1	1	1	1	1	1	0	1	1	0
	N2	2	1	1	1	1	1	1	1	0
P4	N0	0	1	1	1	0	0	1	0	0
	N1	1	1	1	1	1	0	1	1	1
	l	l= l.prev	l=l. next.next	l=l. prev.prev	l=l. next.prev	l=l. prev.next	cnt1= list.size	cnt1$\leq$ list.size	l= list.header	label
F3	N0	0	0	0	1	0	0	1	1	0
	N1	0	1	0	1	1	0	1	0	0
	N2	0	1	0	0	1	0	1	0	0
P4	N0	0	0	0	1	1	0	1	1	-1
	N1	0	1	0	0	0	0	1	0	+1

the correct output. *Our insight* is that considering that branch conditions usually behave uniformly for sets of similar inputs, we could localize the faulty branch occurrences based on the behavior of the branch on similar passing test executions. This localization would not only reduce the search space for path exploration but also help choose the path closest to ideal behavior of the program. Hence, we employ semi-supervised learning enabled by support vector machine, SVM [13], to utilize the spectra of the passing tests to predict the behavior of the branch on the failing tests.

Applying this idea to our example, we intend to obtain the correct behavior for statement 5 on the failing test inputs based on its behavior for the passing test inputs. We build the SVM input by creating one row for every execution of the impacted branch (every occurrence) for both failing and passing tests. Each row is assigned a class label that represents the output of the branch on that input. For passing tests, the output produced by the occurrences of the branch during the actual execution is used (1 for true, -1 for false). For failing tests, the class labels are unknown (0). SVM looks for a classifer (a hyperplane) that distinguishes the

Table 6.5: ListERR1 SVM predictions

Test	cnt1	l	l.next	Predictions	Label
Test-Suite 1					
F1	0	N0	N1	-0.8701	-1
	1	N1	N0	0.7077	+1
F2	0	N0	N1	-1.1623	-1
	1	N1	N2	-0.3441	-1
	2	N2	N0	1	+1
F3	0	N0	N1	-1.1623	-1
	1	N1	N2	-0.0519	-1
	2	N2	N1	1	+1
Test-Suite 2					
F1	0	N0	N1	-0.9999	-1
	1	N1	N0	-0.3762	-1
F2	0	N0	N1	-1.4646	-1
	1	N1	N2	-0.8820	-1
	2	N2	N0	-0.0259	-1

positively labeled rows from the negatively labeled ones, in terms of the attributes or features of the input rows. It uses this classifier to predict the class labels for the failing tests or the behavior of the branch for the failing tests. Table 6.4 shows a sample of the SVM input for the above example. The features comprise of the variables visible at the program point before the branch statement (l, $cnt1$). Please refer Section 6.2.3 for more details on selection and generation of the features.

The predictions generated by SVM are shown in the Table 6.5. We label the failing test rows based on the sign of their respective predictions. For the above example, SVM prediction based labeling assigned 1(true) to those rows where $l.next$ pointed back into the list and -1(false) otherwise. Re-running the failing tests again with the branch condition being forced to behave as labeled for each iteration, we find that all tests pass. Note, however, that in order to make the failing test pass, it would suffice if the condition on statement 5 evaluated to true on some iteration of the while loop. However, similarity based prediction correctly identifies the points where cycles are actually present in the list and thus the correct iteration for which

the branch condition needs to return true. This adds proof to our hypothesis that the correct intended behavior for the failing tests can be guessed based on the behavior of similar passing tests.

Once the correct behavior of the impacted branch has been determined, the next step is to alter the condition such that the expected behavior is produced for both failing and passing tests. The search space for the correct expression can be huge and is proportional to the number of variables and reachable heap states at that program point. Also, for non-pointer variables the number of possible comparisons with absolute values can be huge. Further, we need an expression that is not just correct for the given set of tests but is generalizable. *Our second insight* is to employ data-mining techniques to efficiently generate compact conditions that would be accurate on unseen inputs as well. Decision-tree based classifier learning algorithms, generate classifiers in the form of disjunction of predicates in terms of the features of the input data. The algorithm utilizes the data coverage of a clause to guide the generation of a compact classifier that distinguishes the positive labels from the negative. The most compact classifier can be considered to be the most generalizable and hence would be a useful repair suggestion.

Applying the ID3 algorithm [82] to our example, we obtain the following classifier for label 1 or the condition that makes the branch of statement 5 to evaluate to true, $(l = l.next)||((l.next! = null)\&\&(l! = l.next.prev))$. This condition correctly describes the intended behavior of branch 5 or is the specification for correct behavior at that program point. It can be applied to detect cycles on unseen inputs as well (provided there $prev$ field values are correct). The decision tree

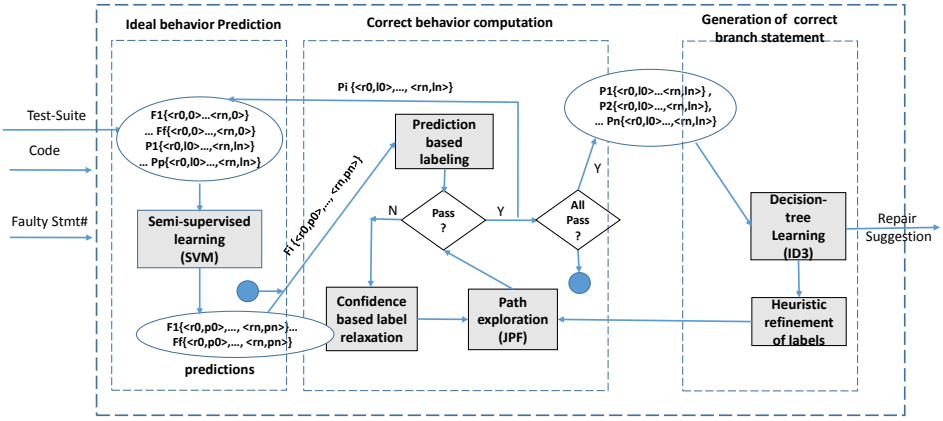

Figure 6.10: Block Diagram.

could be easily translated to an equivalent if-else-if condition in the code, as shown in Listing 2.

Note that this repair involves the generation of a new else branch with an if condition. This would be non-trivial to generate using any other repair technique using mutations or even genetic programming.

```
if (l == l.next){
   res=false;
   break;}
 else
     if ((l.next != null) &&
         (l != l.next.prev)){
         res = false;
         break;}
```

Listing 6.2: ListERR1 repair

6.2.3 Algorithm Details

The block diagram of our algorithm is shown in Figure 6.10. The inputs to the technique along with our assumptions about the problem space are listed below,

- A program with a bug. We assume that the only impact of the fault is on a single branch statement.
- The impacted branch statement. We assume that a mechanism to localize the fault accurately is available.
- A test suite with failing and passing tests, with the assumption that the impacted branch statement has some passing test coverage.

The algorithm comprises of three main modules; i) *Ideal behavior prediction* for the failing tests based on the passing tests, enabled by *SVM* (Section 6.2.3), ii) *Correct behavior computation*, which validates the predicted behavior and if invalid, searches for the correct behavior using *JPF* (Section 6.2.3), iii) *Generation of the correct branch statement*, accomplished by decision-tree learning(Section 6.2.3).

Ideal Behavior Prediction:

This module takes as input, a set of rows corresponding to the failing and passing tests (F_1:$\{< r_0, 0 >, ..., < r_n, 0 >\}$,... F_f:$\{< r_0, 0 >, ..., < r_n, 0 >\}$ and P_1:$\{< r_0, l_0 >, ..., < r_n, l_n >\}$, ..., P_p:$\{< r_0, 0 >, ..., < r_n, 0 >\}$), f and p are the number of failing and passing tests respectively. The row r_i corresponds to the state at the program point before the i^{th} occurrence of the impacted branch statement in the test case execution. Each row has a label associated with it, which

corresponds to the behavior of the faulty branch statement on the respective input state. For passing tests, the label l_i is based on the actual behavior of i^{th} occurrence in the execution, while for failing tests, the labels are 0 or unknown and need to be predicted. Each row is characterized by *feature* variables.

Feature selection: Support Vector Machines (SVM [13]) generate a classifier that distinguishes positively labeled data-points from the negative ones based on the geometric distance between their respective feature values. The failing test rows that are closer to the passing rows in geometric space are predicted to have the same label. When applied to the behavior of a branch statement, this classifier can be considered equivalent to the branch condition. Hence the features need to be variables and/or clauses that could impact the behavior of the branch. They need to be useful and relevant in comparing the state of failing and passing executions at that program point.

The variables of interest (*relevant or base variables*) are determined by short listing those that get defined and/or updated within two edges of the faulty branch statement in the control flow graph. We also consider de-referencing of variables on every field contained in their respective types. Similarly, the addition operation is applied amongst integer variables. User input is also taken (optionally) to refine the relevant variables list, based on his knowledge of the code. For instance, the user could filter out $cnt1$ from being relevant for our example.

Branch conditions usually do not comprise of clauses that use absolute values for pointer variables (such as l). Hence, we include comparison of pointer variables with $null$ and with other variables of the same type. Pointer variables

may have different values for different tests but they may be similar with regards to the comparison clauses. On the other hand, for integer variables like $cnt1$, the absolute values held by these variables can impact the behavior of the branch and hence these variables are directly fed in as features. For integer variables, we generate $=$ and $<=$ comparison clauses with every other integer variable. Global variables and data-structure fields such as $header$ and $size$ may not be in the relevant variables list but are used in the RHS of the comparison clauses. We heuristically assume that the comparison clauses would definitely involve the stack variables, so the LHS is always set to the base variable which are on the stack. For instance, we do not include clauses that compare one de-referenced pointer with another.

It is tricky to determine the number of times to de-reference pointers. On the one hand, the functionality may require considering the structure of the heap beyond the current state on the stack. For instance, in the case of the example, $l = l.next.next$ needs to be detected as a loop. Without this comparison clause as a feature, a failing row where $l! = \ any\ other\ node$ in the list and a row where $l! = l.next$ but $l = l.next.next$ would be considered equally dissimilar with a passing row where $l = l.next$. Inclusion of the feature $l = l.next.next$, would decrease the geometric distance between the passing row and the latter failing row, thereby increasing its similarity. This would enable SVM to predict the latter failing row to have a positive label with higher confidence. On the other hand, when the functionality does not require a global view, inclusion of many de-references may give rise to irrelevant features which may in turn impact the accuracy of the predictions. The default is set to the size of the biggest test input. The user may (optionally) fix this

number explicitly based on his knowledge of the functionality.

Semi-supervised learning using SVM: SVMLight [62] is employed to generate the predictions for the failing test rows (Table 6.5). The sign of the prediction represents the label and the absolute value of the prediction is the confidence of prediction.

Correct behavior computation:

This module validates if the predicted behavior for the faulty branch occurrences yields the correct final output and if not, employs JPF to search for path/s that produce the correct result. Correct behavior computation is done individually for every failing test. The code is instrumented such that the faulty branch behaves as per the predicted labels and the test is executed again. For the list example, for all the failing tests, the predicted labels yield the correct output. For instance for F1, in the second iteration of the while loop when $l = N1$, $N1.next$ points to $N0$, and so the prediction that the condition to detect the loop should evaluate to true is correct.

Incorrect Predictions: Let us consider the same example with a different test-suite where test case P1 is a three node list without loops (Test-suite2 in Figure 6.9). In this case, for tests F1 and F2, all the rows are predicted to have negative labels (Table 6.5). This implies the condition does not detect loops in any iteration and so these 2 tests fail. Although the passing test coverage is the same as before, the diversity of passing behavior is not good and hence the predictions are skewed in the direction of the negative labels. In such cases, we take guidance from the

confidence values to guess which of the predictions may be wrong. The prediction with the lowest confidence value for F1 is for the 2^{nd} iteration (0.3762) and for F2 it is the 3^{rd} iteration (0.0259). The predicted labels are relaxed on these rows and the respective tests executed again with *JPF* being used to non-deterministically generate the output (true and false) for the branch in the respective iterations. The paths generated by each of the labels is explored to determine if the final output yields the correct result. Both F1 and F2 pass when the branch evaluates to true in the 2^{nd} and 3^{rd} iterations respectively. Hence the labels for these rows are marked +1.

In case, the correct output cannot be obtained even after relaxing the labels on the lowest confidence rows, the algorithm looks for the prediction with the next lowest confidence value and sets it as a new *confidence threshold* for label relaxation. This process of iterative relaxation of labels and JPF based exploration of the possible paths continues until the test passes.

Multiple solutions: There could be situations where the JPF detects more than one paths making a test pass. For instance, let us assume that the predictions generated using test-suite 2 were the same for both the iterations for F1 (-0.3762). With the confidence threshold set to 0.3762, the labels on both these iterations would get relaxed, which would lead to two possible labelings that make the test pass (-1,-1,+1) and (-1,+1,NA) for the the three iterations respectively. Both these solutions are considered equally probable and passed on to the subsequent module.

Please note the following points about the execution of a branch based on labeling or non-deterministic choice.

1. Forcing a particular occurrence of a branch statement to behave as per a label or a non-deterministic choice may invalidate other occurrences which were part of the original failing execution. For instance, if the branch in our example (which is within a loop) is forced to evaluate to true in an iteration, the occurrences for subsequent iterations (if any) would not be applicable. Similarly, occurrences with input states not encountered in the original execution may also appear. For any such new input, the behavior is determined non-deterministically.

2. The branch may be executed on the same input state more than once. We set a time-out limit to prevent infinite loops.

3. The branch may behave differently for the same input, when the behavior is determined non-deterministically. As a post-processing step, we invalidate such solutions.

As mentioned earlier, the validation of predictions and JPF based path exploration is done separately for each failing test. Once a test passes, the updated set of rows and correct labels are fed back into the SVM-based learning module. For tests, where the JPF-based path exploration time or the number of possible solutions exceed a threshold limit, predictions are regenerated based on the updated SVM input. The new predictions are likely to be more accurate and aid in reducing the search space.

Generation of the correct branch condition.

This module takes as input the set of rows corresponding to all the tests in the suite and the respective labels that make them pass, (P_1:{$< r_0, l_0 >, ..., < r_n, l_n >$}, ..., P_n:{$< r_0, l_0 >, ..., < r_n, l_0 >$}). Note that l_i is assigned a value 1 for a positive label and 0 for a negative label.

Decision-tree learning: We apply a home-grown implementation of the ID3 decision-tree learning algorithm [82] to generate a classifier that generates the labels. The ID3 algorithm repeatedly splits the input rows into smaller groups by performing a test on an attribute such that each group maximally contains labels of one particular type. This is continued until all groups contain labels of only one particular type. This process builds a a decision-tree and the disjunction of paths (attribute tests) corresponding to the groups of one particular type form the classifier for that type. This classifier is in the form of disjunction of conjunctions of clauses which is the syntax of branch conditions in imperative programs.

In order to generate a compact decision-tree and classifier, at each level, the algorithm chooses the attribute test which provides the maximum information gain or reduction in entropy to perform the split. The resultant classifier is the smallest or the most compact rule that classifies the respective input correctly according to the labels. A compact classifier is usually generalizable and could be considered closest to the ideal oracular fix.

Attribute selection: The features used for SVM based learning are fed as attributes to the decision-tree learning algorithm as well. Conditions involving local pointer variables usually do not involve more than 1 field de-references. However, this may be required for the repair depending on the type of the error. For instance, a faulty

branch behavior within a loop, could lead to an accumulation of faulty state in such a way that more than 1 de-references may be required to obtain the correct state.

Avoiding Overfitted classifiers: Note that our implementation of ID3 does not perform any pruning and has a zero error threshold. This is because we want to produce a classifier that is 100% accurate for the tests which act as the training data, so that they pass. In order to avoid over-fitting of the classifier to the tests and ensure that it is generalizable, we make the following amendments to the algorithm based on heuristics.

1. When more than one attributes have the same entropy, we rank them such that attributes which involve clauses with lesser number of field de-references are given more priority than others. The rationale being that more compact expressions tend to be more globally valid.

2. Despite the first amendment, classifiers with considerably large number of clauses and field de-references may be produced. This may be the requirement of the repair or this could be over-fitting. Over-fitting may happen because the paths, represented by the failing and passing rows and labels, may produce the correct result for the test-suite but may not be the ideal paths. In order to avert this, as a post-processing step the algorithm attempts to remove clauses with expressions having more than a *threshold* number of de-references. This may invalidate the labels on certain rows. JPF is invoked again to determine if the new labels can generate alternate paths that produce correct outputs. If so, then learning is invoked again using the new labels to

determine if a more compact decision-tree can be learnt. This is referred to as *Heuristic refinement of labels* in Figure 6.10.

3. Over-fitting could also lead to classifiers that are shorter than ideal, i.e. it may miss some clauses that are redundant for the given test-suite but need to be included for global validity. As another post-processing step, the algorithm checks if any of the clauses already present in the original condition is redundant for the inputs in the presence of the learnt classifier. Such clauses are appended to the classifier thus reducing the distance between the original and the repaired condition.

Multiple labelings: When there are multiple possible labellings or multiple solutions produced by the previous module for failing tests, sets of inputs are generated , each comprising of a particular combination of solutions for the different failing test merged with the set of passing rows. The classifiers generated for all the input sets are ranked based in the ascending order of compactness (# of clauses and # of field de-references). The repair suggestions are presented to the user in the ranking order.

Other errors impacting branch behavior:

Variable update errors: Our algorithm can also handle erroneous variable updates that may impact the behavior of a branch. However, we require the faulty statement to be accurately localized. Further, the erroneous update should only impact the behavior of a branch such that path exploration based on that branch's behavior can produce the correct final output.

We retain the variable as a feature during SVM prediction, despite its update being erroneous. The rationale being that since the branch does work for passing tests, failing tests having similar value for the variable could be assumed to have correct behavior. However, we also include as features, the variables that the update is a function of, and the possible operations that could be used in the function. Dissimilarity introduced by these variables would help differentiate or isolate the defect-inducing rows of the failing test. We also retain the impacted variable in the decision-tree learning phase since we want to have minimal changes from the original condition (which uses the variable).

The condition learnt by applying our algorithm is equivalent to a *specification of correctness* at that program point. Any specification based repair or synthesis technique may then be applied to obtain the correct repair for the variable update.

Loop construct errors: Our algorithm can identify and repair erroneous *while* constructs which should ideally have been *if* conditions (*while-to-if*). In the process of heuristic refinement of labels, our algorithm makes a special check when the impacted branch is a while construct. If no alternate paths can be detected based on the new labels, the algorithm checks if the correct output can be obtained by not considering any of the rows with invalidated labels (i.e. JPF would not explore any paths for those states). If this is possible, it generates a classifier based on this reduced set and considers it as the condition for an *if* statement instead of the original while construct.

It is difficult to correct the bug if it is the other way around *if-to-while*. The reason being that this would involve force executing the branch on new inputs, by

Table 6.6: Faults

Fault#	Fault	Generated Repair(GR)	Oracular Fix(OF)	GR ≡ OF
ListERR1	if (l == l.next){ ... }	if (l == l.next){ ... } else if ((l.next != null) && (l != l.next.prev)){ ... }	same as repair	Y
ListERR2	while(cnt2 < size)	while((cnt2 <= size) && (p != null))	same as repair	Y
ListERR3	p = l.next	p = l.next ... if(l == p){ ... } else if((l.next != null) && (l != p.prev)){ ... }	p = p.next	Y
RBTERR1	while((p != null) && (p.left == null))	while((p != null) && (ch == p.right))	same as repair	Y
RBTERR2	while(p != null)	while((p != null) && (ch != p.right))	while((p != null) && (ch == p.left))	N
RBTERR3	ch = e.parent.right	ch = e.parent.right ... if (e == p.right)	ch = e	Y
RBTERR4	if (e.right != null) && (e.left != null)	if (e.right != null)	same as repair*	Y
BSTERR1	while((x != null) && (k < x.key))	if((x != null) && (k < x.key))	same as repair*	Y

transferring control back to the program point before the branch statement. JPF, an explicit state model checker, cannot execute an if condition as a while loop. However, it can be made to ignore paths corresponding to certain iterations of a while loop, which enables us to handle *while-to-if* repairs.

6.2.4 Evaluation

The aim of our evaluation was to determine the efficacy of our approach to correct different types of faults that impact the behavior of a branch in a program. We also propose to address the following research questions,

RQ1: How effectively can the correct behavior of failing tests be predicted based on the behavior of similar passing tests?

RQ2: How accurate is the repair generated by decision-tree learning?

Subjects: We chose three data-structure programs to evaluate our approach. These programs have non-trivial functionality making it challenging to correct bugs manually or using other mutation or search based techniques, specifically in the absence of correctness specifications. Further, they mostly handle heap-allocated structures and hence number of reachable states at any program point is considerably large. The program functionality is mostly recursive in nature. Hence the faulty branch usually has high coverage, however the bug manifests itself only on some occurrences. It is challenging to make a repair that not only fixes the code for given tests but is generalizable and close to an ideal oracular fix. These characteristics may these subjects ideal candidates for the evaluation of our learning based approach.

- **List.repOK:** This is a commonly used method which checks if the underlying doubly linked list (DLL) structure complies to its invariants (acyclicity and size check). We consider two different versions of this method (repOK (Listing 1) and repOKmod (Listing 3).

- **BST.insert:** Binary Search Tree(BST) is a popular data-structure used in search algorithms. The structure is fairly complex, with each node having at most two children (left and right) and pointer to its parent. The class invariants are; acyclicity of pointer fields, uniqueness of key, and search constraints (key of every node is larger than the keys in its left sub-tree and smaller than

the keys in the right sub-tree). The code for the **insert** method is shown in Listing 5.

- **java.util.TreeMap.containsValue(Object):** Java.utils is a very commonly used library. The TreeMap class has a Red-Black Tree implementation, which is one of the most complex data-structures in terms of the invariant constraints. The **containsValue** method returns true if the map maps one or more keys to the input value.

Faults: We manually seeded faults into the subject methods (Listings 3,4,5). As can be seen, we simulated both omission and commission errors, incorrect and/or missing clauses, fault in operators and loop constructs, and faulty variable updates. Only one of the faults were present in the code at a time (For instance, ListERR2 and ListERR3 were not simultaneously present in List.repOKmod). We ran our algorithm for every fault in each subject individually. We used randomly generated test-suites; List.repOK(8 tests), List.repOKmod(9 tests), TreeMap.containsValue(11 tests), and BST.insert(8 tests) respectively.

Our implementation uses SVM Light [62] in transductive learning mode, Java Path Finder model checker version 6.0, and a home-grown implementation of the ID3 decision-tree learning algorithm. All experiments were conducted in a 2.53Ghz CPU, 4GB RAM laptop running Windows 7.0.

Table 6.7 shows the statistics for the SVM predictions-based labeling and correct behavior generation modules. **Passing test coverage** shows the % of SVM input rows that belonged to the passing tests.

Table 6.7: Results

Fault#	Passing test coverage(%)	Predictions based labeling		Reduction in JPF search space(%)
		Recall(%)	Accuracy(%)	
ListERR1	53(9/17)	100	100	100
ListERR2	41(18/43)	84(21/25)	47(10/21)	46.7
ListERR3	53(17/32)	100(6/6)	100(6/6)	100
RBTERR1	60(12/20)	53(8/15)	87.5(7/8)	91.6
RBTERR2	18.5(5/27)	60(12/20)	41.6(5/12)	3.6
RBTERR3	40(13/32)	42(8/19)	62.5(5/8)	94.2
RBTERR4	45(14/31)	100(7/7)	57(4/7)	81
BSTERR1	68.4(13/19)	50(4/8)	50(2/4)	25

Metrics: We evaluated the following metrics for the application of our algorithm on each fault.

Recall(%): This metric calculates the percentage of rows in the correct solution, that were also part of the original execution.

Recall = $\#r_i$ s.t. r_i in $\sum_{1}^{f}(\{rin_0, ..., rin_{nin}\} \cap \{ro_0, ..., ro_{nop}\}) / \sum_{1}^{f} \#\{ro_0, ..., ro_{nop}\}$, where in represents the rows in the SVM input, o represent the rows present in the actual solution and nin, nop represent the respective set sizes. Let us term these rows and their respective predictions *relevant*.

Prediction Accuracy(%): This metric determines how many of the relevant predictions were accurate or assigned the correct label to the respective rows. Accu-

racy = $\#r_i^{relevant}$ s.t. (sign of p_i) = l_i) / $\#\{r_i^{relevant}, ..., r_n^{relevant}\}$, where p_i, l_i are the predictions and labels for the i^{th} relevant row respectively.

Reduction in search space(%): In the absence of any predictions, we could solely use JPF based path exploration to determine path/s that produce the correct solution. The search space in such a case would in the worst case be 2^b, where b represents the number of occurrences of the faulty branch. With the help of predictions, we fix the behavior of the branch on certain inputs and hence reduce the space of paths to be explored. This metric measures the % reduction as $\#(b - b_{pred})/\#b$, where $\#b_{pred}$ represents the number of branch occurrences for which the predicted label produces the correct behavior.

Table 6.6 shows the repairs generated by our approach for each fault (GR). Oracular fix represents the original statement in the code which is considered the ideal repair (OF). We tabulate if GR is syntactically same as OF and also if GR $\equiv$ OF. The latter indicates behavioral or semantic equivalence for all inputs.

Case Studies:

We first describe the application of our approach on specific examples to highlight the design decisions and intricacies of our algorithm.

Scenario 1: Near accurate predictions and ideal repair: This scenario highlights the ability of our algorithm to efficiently generate an accurate repair for a branch condition with an incorrect clause in a while condition (RBTERR1 in Listing 4).

The *TreeMap.containsValue* method (Listing 4) parses the tree (using the variable e) by starting from the first entry which is the leftmost child of the $root$.

Subsequently, while $e! = null$, its value is compared with the input. If a match is found, true is returned. If not, the successor is determined, which is the left most child of $e.right$. If $e.right$ is null, then the tree is parsed upwards until e is not the right child of its parent, at which point the variable e is set to its parent's value and the comparison with input is made again. This process continues until $e == null$ and false is returned if a match has not been found until then. RBTERR1 corrupts the parsing logic when $e.right == null$. It is non-trivial to manually correct subtle bugs in a complex code such as this.

Generation of correct behavior: The base variables for the features are the pointer variables e, p, ch and their de-references for fields $left$, $right$ and $parent$. the number of de-references is set to 3 (depth of the biggest input). There were 477 total features. Note that this is equivalent to the population of variants for the condition at that program point. It would be infeasible for a technique based on search in the program space since it would also need to look at combinations of two or more expressions from the population, joined by operators such as OR and AND.

The coverage of the passing tests was high (60%) with good behavioral diversity which aided in the generation of near accurate predictions. Labeling based on predictions makes 2 failing tests pass, while the third one passes by changing the predicted label on just one row. This resulted in a search space saving of 91.6% if JPF-based path exploration were employed without any predicted labels.

Generation of correct condition: The decision-tree learnt was exactly same as the ideal condition $(p! = null)\&\&(ch == p.right)$. Note that the condi-

tion ($p! = null$) was included despite being redundant (was true for all rows where ($ch == p.right$)).

Scenario 3: Variable update and loop construct faults: This scenario demonstrates the application of our algorithm on faults which indirectly impact branch behavior.

ListERR3 in repOKmod (Listing 3) is an erroneous update to the local variable p which in turn impacts the behavior of the branch on statement 8. Only input lists which have self-loops are detected as cycles, similar to the behavior of the repOK method with ListERR1 as the fault.

Given that variable p impacts only the branch at statement 8, the correct output for the program can be obtained by exploring the paths through the branch. We determined that the update would be a function of l,p and de-references of $left$, $right$ and $parent$. These were included as the feature variables.

The decision-tree learnt was $(l = p)||((l.next! = null)\&\&(l! = p.prev))$. This is not syntactically same as the repair of changing $p = l.next$ to $p = p.next$ on statement 12. However, it is functionally or behaviorally equivalent since both the repaired versions would provide the same output for any input (provided the $prev$ field is set correctly). The decision-tree actually serves as a globally valid ***specification for a cycle*** at the program point before statement 8. Any mutation or synthesis based technique could then be used to look for expressions for p (other than $l.next$) at statement 12 that would satisfy this specification.

Consider the bug **BSTERR1** in the BST.insert code (Listing 5). The while

loop at statement 3 parses through the binary search tree abiding by the search constraints. Due to the bug on statement 7, as long as the input $k < x.key$, x is constantly updated with $x.left$ until it becomes null. This leads to failures where the variable y does not contain the correct point where the insertion needs to be made. Based on the initial round of correct label computation, the decision-tree generated was $while((x! = null)\&\&(k < y.key)\&\&(x == y))$. Heuristic refinement was invoked on the condition, in an attempt to make it more compact. On removing the clause $(x == y)$, the labels on all rows (except those corresponding to the first iteration of the while loop) were invalidated for all the failing tests. Since x gets updated within the loop and y does not, the new condition marks their labels to be positive which differs from the previous labeling. JPF is unable to make the tests pass with the new labels, however, the tests pass when these rows are skipped from the execution of the while loop. The condition learnt from the shortened set of rows is $if((x! = null)\&\&(k < x.key))$ which is the desired repair.

Scenario 2: Impact of poor passing test coverage: This scenario demonstrates how our algorithm handles the impact of poor passing test coverage on predictions.

The bug ListERR2 in the while condition in List.repOkmod (Listing 3) inhibits the detection of cycles leading to the failure of 6 out of 9 tests. None of the passing tests possess cycles and hence are quite dissimilar to the failing tests. This leads to poor prediction accuracy (47%), which increases the search space for JPF-based path exploration. Specifically for one of the tests, the row which had to ideally have a positive label was assigned a prediction of -1.68. This confidence value was the largest amongst all the predictions for the test and hence in order

to obtain the correct solution, the labels for all the rows had to be determined by JPF based exploration. The number of possible solutions were also huge ranging from the smallest solution set containing just 1 row for every value of l, $\{< cnt1 = 0, cnt2 = 1, l = N0, p = N1, label = -1 >, < cnt1 = 1, cnt2 = 2, l = N1, p = N2, label = -1 >, < cnt1 = 2, cnt2 = 3, l = N2, p = N2, label = 1 >\}$ to the largest set with the same row for l=N2 but 3 and 2 rows for $l = N0$ and $l = N1$ respectively, with -1 only assigned to the row where $p = NA$. We invoked SVM to predict the labels for this again based on the updated labels for the other tests. The new predictions pointed to just 1 path (the largest solution set) which is the ideally desired behavior.

Discussion:

This section addresses the research questions based on the results presented in tables, Table 6.6, and Table 6.7.

RQ1: The effectiveness of similarity based prediction of correct behavior for failing tests is dependent on the passing test coverage. In most cases, where the passing coverage is $> 50\%$, the prediction accuracy is $> 80\%$. The diversity of the behavior of the branch for the passing tests also impacts the accuracy, otherwise the predictions tend to get skewed towards one label.

Low recall can also impact the quality of predictions as is the case with BSTERR1. Due to less number of *relevant* rows in the SVM input, data points used for semi-supervised learning are not representative enough, leading to poor predictions. Further, the logic of the incorrect code was such that the behavior of

the branch was not same as the ideal, even for some of the passing tests but there existed an alternate path that produced the correct output for these tests. The wrong labels for the passing tests impacts the prediction accuracy for the failing tests.

RQ2: In all the cases, the generated repair was 100% accurate for the test-suite. In 5 out of 9 cases, the repair generated was exactly the same as the ideal fix. For the cases marked with *, the initial decision-tree was overfitted (with number of clauses and de-references > 2), however the ideal condition was arrived at by heuristic refinement of labels. In the case of RBTERR2, there were no input instances where $ch! = p.left$ and $ch! = p.right$, hence due to the lack of representativeness of the test inputs, the generated repair is not globally valid. For faults in local variable updates, the algorithm generates a specification, which in most cases contains hints to the repair of the update statement. Note that the repair generated for RBTERR3 is $if(e == p.right)$. It can be determined by simple program analysis that at that program point $p = e.parent$ and ch has been incorrectly set to $p.right$. This indicates that setting ch to e should produce the same effect as the if condition.

6.2.5 Related Work

This section compares our work with other recently proposed approaches for program repair.

Use of machine learning for program repair:

Our previous work [43] employs machine learning to repair incorrect *Where clauses* in database statements in ABAP programs. In this paper, we have extended the idea to repair branch conditions in imperative programs. Although the crux of

the approach is similar, there are couple of differences between the two domains. Input to database statements are tables with rows characterised by attributes, which is exactly the form of input for SVM. Each row is identified by a key field and in a single execution the selection statement may behave incorrectly for certain keys (failing keys) and correctly for others (passing keys). Each row is independent of the others. In an imperative program, on the other hand, the mapping of program state to SVM input format is not straight-forward. Each row represents different occurrences of the branch condition during a single execution and hence they may be dependent on each other. This necessitates the use of JPF to explicitly execute the code to explore subsequent states. The output of a selection statement is always a subset of its input rows and a SAT-based combinatorial search is employed in the previous work.

Our work differs from Alijareh et.al [6], which employs logic learning to diagnose and correct errors in finite state transition systems, in the following ways. ILP can express first order logic constraints and can probably generate programs with complex logic. However, a fully specified property needs to be provided. Our approach, on the other hand, works with just the test-suite and attempts to generate a globally valid condition. We focus on incorrect branch behavior, for which decision-tree learning can quickly generate correct solutions. We assume accurate fault localization and restrict the region of repair locally around the faulty statement. We attempt to learn not a global property of correctness but the condition for a specific branch functionality. Such a specification can be expressible as disjunctions of conjunctions and would not involve first order logic operations.

The Daikon tool based data-structure program repair described in [31], learns specification constraints from passing test behavior alone. Hence they represent general data-structure constraints which may be different from the specification of correctness at the point of fault. Therefore, the repaired data-structure may satisfy the data-structure invariants but may not be the same as the output intended by the ideal behavior of the program. Further, the technique is only applicable to programs that mutate the data-structure.

Chapter 7

Conclusion

Debugging, i.e., locating and removing bugs in faulty code, is expensive and itself error-prone. We introuduced a suite of techniques for more effective debugging. Our key insight is that integrating systematic search based on state-of-the-art constraint solvers with techniques to analyze artifacts (such as correctness specifications and test cases) that describe application specific properties and behaviors provides the basis for developing more effective debugging techniques. We focused on faults in programs that operate on structurally complex inputs, such as heap-allocated data or relational databases. Debugging such faults using traditional methods is particularly ineffective and time consuming. We used a number of real-world subjects to evaluate our techniques and show their effectiveness. We believe our work provides a viable foundation for new debugging approaches that further improve the effectiveness of automated debugging while lowering its cost.

Appendices

Appendix A

Background Information

A.1 Alloy constraints for SLL

This section describes the Alloy model of the singly linked list data structure (Figure 2.1 in more detail. `SLL` and `Node` are signatures of the respective Java classes. Fields of these classes such as `header`, `size`, `next` are represented as binary functions. The multiplicity of the functions are expressed using keywords such as `lone`, `some`, and `no`. For instance, the `lone` keyword indicates that `header`, and `next` are partial functions, while `size` and `next` are total functions. The `pred repOk` defines the invariant constraints on the linked list, that is passed on to it as parameter `l`. It consists of three constraints; `acyclicty`, `size-invariant` and `unique-elements`. The `acyclicity` constraint ensures that for every node in the list, it should not be possible to reach the same node , when parsing through the subsequent nodes via the `next` pointer. The join `l.header` accesses the node mapped to the list present in the scalar `l` by the `header` relation. Operators * and represent reflexive and transitive closures on relations. For instance, `*next` represents the relations `next`, `next.next`, `next.next.next`, so on until the operation results in nulls. Hence, `l.header.*next` accesses all the nodes reachable from the `header` node until null is hit. The universal quantifier `all` indicates that this constraint applies to all the elements in the set accessed by `l.header.*next`.

Cardinality, membership and complement operations are represented using the keywords; #, `in` and – respectively. Alloy also supports logical binary operators, such as implication ($\Rightarrow$ in uniqueness constraint).

A.2 Code to Logic

We illustrate in detail how the `delete` procedure is translated to a formula in relational logic $P(s, s')$.

```
boolean delete(int k){
 0:      Node prev = null;
 1:      Node lst = header;
 2:      if(lst != null){
 3:         if(lst.key == k){
 4:            if(prev != null){
 5:               prev.next = lst.next;
                  }else{
 6:               header = header.next;
                  }
 7:            size--;
 8:            return true;}
 9:         prev = lst;
10:         lst = lst.next;
         }
11:      if (lst != null) {
12:         if(lst.key == k){
13:          if(prev != null){
14:            prev.next = lst.next;
               }else{
15:             header = header.next;
               }
16:         size--;
17:         return true;}
18:         prev = lst;
19:         lst = lst.next;
         }
20:      assume(lst == null);
21:       return false;}
```

Listing A.1: SLL delete method with 2 loop unrolls.

Every loop in the control-flow graph of the procedure is unrolled a predefined number of times and is represented as a nested-if statement. Listing A.1 shows the computation graph after unrolling the `delete` procedure two times.

Symbolic execution is employed to convert the FIR representation of the procedure to relational logic.

The state of the symbolic execution at each program point has three components; i) *Relational declaration* of the set of relations at that program point(D), ii) *Path Constraints*, set of constraints on the relations that need to be satisfied for execution to reach that program point(P), iii) *Environment*, mapping the variables and fields in the program to the respective relations in logic(E)). For instance, the initial symbolic state of the `delete` procedure would be as shown below,

D_{init} = $List0$,$Node0$,$size0$,$header0$,$next0$,$prev0$,$key0$,$k0$

P_{init} = $(header0 \subseteq List0 \rightarrow Node0)$, $(next0 \subseteq Node0 \rightarrow Node0)$, $(prev0 \subseteq Node0 \rightarrow Node0)$, $(size0 \subseteq List0 \rightarrow Integer)$, $(key0 \subseteq Node0 \rightarrow Integer)$

E_{init} = $(This \mapsto List0)$, $(Node \mapsto Node0)$, $(size \mapsto size0)$, $(header \mapsto header0)$, $(next \mapsto next0)$, $(prev \mapsto prev0)$, $(key \mapsto key0)$, $(k \mapsto k0)$

The following binding satisfies the path constraints in P_{init}; $List0 = \{L0\}$, $Node0 = \{N0, N1\}$, $size0 = \{1\}$, $header0 = \{< L0, N0 >\}$, $next0 = \{< N0, N1 >\}$, $prev0 = \{< N1, N0 >\}$, $key0 = \{< N0, 1 >, < N1, 2 >\}$, $k0 = \{2\}$. However, it does not represent a valid input state that satisfies the size constraint, part of the `repOk` invariant. Such invalid bindings/states would be eliminated when the formulation for SAT includes an invariant check on the pre-state in conjunction with the constraints corresponding to the code, ($Pre - condition(s) \bigwedge P(s, s')$).

The components of the symbolic state is updated after the execution of each statement. For instance, after the assignment statement 0 `Node prev = null`, $D0 = D_{init} \bigcup prev0$, $P0 = P_{init} \bigcup prev0 = \{\}$ and $E0 = E_{init} \bigcup (prev \mapsto prev0)$. Field updates are modeled as relational overrides. For instance, statement 5 `prev.next = lst.next` would be modeled as an override of the `next` relation, wherein tuples with `prev` as the first element would be replaced with $< prev, lst.next >$. The if-then-else code snippet from statements 4 to 6, would update the path constraints as follows $P5 = P3 \bigcup ((prev0! = null) \Rightarrow (next1 = next0 + +(prev0 \rightarrow lst0.next0)))$, $P6 = P3 \bigcup ((prev0 = null) \Rightarrow (header1 = header0 + +(List0 \rightarrow List0.header0.next0)))$. The statements in the then portion of the if statement are added as the consequents of an implication with the if-condition as the antecedent, while the statements in the else portion have the negation of the if-condition as the antecedent. The relational declarations $D5$ and $D6$ are updated as $D3 \bigcup next1$ and $D3 \bigcup header1$ respectively, and the environments $E5$ and $E6$ are updated as $E3 \bigcup (next \mapsto next1)$ and $E3 \bigcup (header \mapsto header1)$ respectively. After the execution of the last statement of the procedure, the final version of the relations have their names suffixed with the back tick ‘.

The symbolic execution finally generates a final symbolic state; D_{final}, including all the declared relations, P_{final}, the path constraints corresponding to all the execution paths in the procedure within the scope of the unrolls, and E_{final}, the final environment. An assignment to all the relations in D_{final} that satisfy the constraints in P_{final} correspond to a valid execution of the procedure and the satisfaction of the formula $P(s, s‘)$. In the specification, $S(s, s‘)$, s corresponds to the

relations in the initial state D_{init}, while s‘ corresponds to the post-state relations in D_{final}. For instance, in the analysis of the `delete` procedure, the `remove-ok` constraint would be evaluated as `List0.header0.*next0.key0-k0= List0.header`.*next`.key`

Appendix B

Code Snippets

```
bool insert(Tree t, int k){
  Node y = null;
  Node x = t.root;
  while (x != null)
  //while(x.left != null) (4c)
  { y = x;
    //y = t.root; (3b)(8)(9)
    if (k < x.key)
    //if(k < t.root.key) (9)(13)
      x = x.left;
      //x = x.right; (3a)
      //x = x.left.right (13)
    else
    { if (k > x.key)
      //if (k < x.key)      (4a)
      //if(k > t.root.key) (4b)
        x = x.right;
      else
        return false;}}
  x = new Node();
  x.key = k;
  if (y == null)
  //if(x != null) (10)
    t.root = x;
  else
  { if (k < y.key)
    //if(k > y.key) (2a)
    //if(k < x.key) (2b)(5)(11)(12)
      y.left = x;
      //y.left = y; (6)(10)(11)
    else
      y.right = x;}
      //y.right = y;}(10)(12)
  x.parent = y;
  //x.parent = x; (1)(7)(8)
  //y.parent = x; (5)(6)(10)
  /*x.parent = y;*/ //Omission Err(14)
  t.size += 1;
  return true;}
```

(a)

```
void addChild(Tree t){
  if ( t == null} return;
  BaseTree childTree = (BaseTree)t;
  if (childTree.isNil()) {
    if (this.children != null &&
        this.children == childTree.children)
      throw new RuntimeException("...");
    if (childTree.children != null) {
      int n = childTree.children.size();
      for (int i = 0; i < n; i++) {
      //for (int i = 0,j = 0; i < n; (i = j + 1)) { (1)
        Tree c = childTree.children.get(i);//list get
        this.children.add(c);//list add
        c.setParent(this);
        //j = i + 1; (1)}}
    else this.children = childTree.children;}
  else {
    if (children == null)
    //if (childTree.children == null) (2)
        children = createChildrenList();
    children.add(t); //list add
    childTree.setParent(this);
    //childTree.setParent(childTree); (2)}}
```

(b)

Figure B.1: Code Snippets used in Section 5.4. Repaired version produced in the CFG form, manually mapped back to source code.

```
boolean add(int k){
0  Node y = null;
1  Node x = this.root;
2  while (x != null)
3  { y = x;
4    if (k &lt; x.key)
5      x = x.left;
    //BSTtraverseErr
       x = x.right;
     else
6    { if (k &gt; x.key)
7        x = x.right;
       else
8        return false;}}
9   x = new Node();
10  x.key = k;
11  if (y == null)
12    this.root = x;
    //BSTrootErr
      this.root = null;
    else
13  {if (k &lt; y.key)
    //BSTbranchErr
      if (k &lt; y.key)
14      y.left = x;
    else
15      y.right = x;}
       //BSTrightErr
         y.right = y;}
16  x.parent = y;
    //BSTparentErr
      x.parent = x;
17  this.size += 1;
    //BSTsizeErr1,2
      this.size = 1;
18  return true;}
```

(a)

```
void addChild(Tree t){
0 if ( t == null} return;
1 BaseTree childTree =
              (BaseTree)t;
2 if (childTree.isNil()) {
3   if (this.children != null &&
      this.children ==
      childTree.children)
4     throw
      new RuntimeException();
5   if (childTree.children
       != null){
6    int n =
     childTree.children.size();
7    if (children != null) {
8    for (int i = 0; i &lt; n; i++)
9    { Tree c =
      childTree.children.get(i);
10   this.children.add(c);
11   c.setParent(this);
      //ANTchildparErr1,2
      c.setParent(c);}}
      else{
12    children =
      childTree.children
13    int i = 0;
14    while (i &lt; n){
15    Tree c =
      childTree.children.get(i);
16    c.setParent(this);
      //ANTchildErr
      c.setParent(c);
17    i++;
      //ANTloopErr
      i = i + 2;}}}}
    else {
18  if (children == null)
19   children = createList();
20   children.add(t);
21   childTree.setParent(this);
     //ANTparentErr1,2
     childTree.setParent;
              (childTree);}}
```

(b)

Figure B.2: Code Snippets used in Section 4.4.

Bibliography

[1] JTopas. `http://jtopas.sourceforge.net/jtopas`.

[2] Software-artifact Infrastructure Repository. `http://sir.unl.edu/content/bios/jtopas.php`.

[3] Ieee standard glossary of software engineering terminology. 1990. IEEE Std 610.12-1990.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *PRDC*, pages 39–46, 2006.

[5] T. Ackling, B. Alexander, and I. Grunert. Evolving patches for software repair. In *GECCO*, pages 1427–1434, 2011.

[6] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Automated support for diagnosis and repair. *Commun. ACM*, 58(2):65–72, 2015.

[7] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "Small Scope Hypothesis". Technical report, MIT CSAIL, 2003.

[8] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, 2010.

[9] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *ICSE*, 2010.

[10] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2003.

[11] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, 2003.

[12] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. 29th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 2002.

[13] K. P. Bennett and C. Campbell. Support vector machines: hype or hallelujah? *SIGKDD Explor. Newsl.*, 2(2):1–13, Dec. 2000.

[14] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel. Theoretical foundations of dynamic program slicing. *Theor. Comput. Sci.*, 360(1-3):23–41, 2006.

[15] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[16] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[17] L. C. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with Tarantula. In *ISSRE*, pages 137–146, 2007.

[18] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *ICSE*, pages 121–130, 2011.

[19] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, June 2002.

[20] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP*, 2002.

[21] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, pages 34–44, 2009.

[22] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[23] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proc. Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

[24] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. 10th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2004.

[25] E. M. Clarke and D. Kroening. Tutorial: Software model checking. In *ICFEM*, pages 9–10, 2004.

[26] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ansi-c programs using sat. *Formal Methods in System Design*, 25(2-3), 2004.

[27] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[28] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. pages 191–195, 1989.

[29] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, pages 65–74, 2010.

[30] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA*, pages 30–39, 2014.

[31] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244, 2006.

[32] G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with sat. In *ISSTA*, pages 109–120, 2006.

[33] G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA*, 2006.

[34] G. D. Dennis. *A Relational Framework for Bounded Program Verification*. PhD thesis, Massachusetts Institute of Technology, 2009.

[35] N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *SAT*, 2006.

[36] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, 2005.

[37] N. Een and N. Sorensson. An extensible SAT-solver. In *Proc. 6th Conference on Theory and Applications of Satisfiability Testing (SAT)*, Santa Margherita Ligure, Italy, 2003.

[38] B. Elkarablieh and S. Khurshid. Juzi: A tool for repairing complex data structures. In *Proc. 30th International Conference on Software Engineering (ICSE)*, Leipzig , Germany, May 2008. Research Demo Paper.

[39] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[40] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen. Codehint: dynamic and interactive synthesis of code snippets. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 653–663, 2014.

[41] E. Gamma and K. Beck. JUnit: A cook's tour. `http://www.junit.org`.

[42] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[43] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-guided repair of selection statements. In *ICSE*, 2014.

[44] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-Guided Repair of Selection Statements. Technical report, IBM Research. India, 2014. IBM Technical Report RI14004, available from http://domino.watson.ibm.com/library/CyberDig.nsf/ho

[45] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using sat. In *TACAS*, 2011.

[46] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *TACAS*, pages 173–188, Mar. 2011.

[47] D. Gopinath, R. N. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *ASE*, 2012.

[48] D. Gopinath, R. N. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *ASE*, pages 40–49, 2012.

[49] C. L. Goues, K. R. M. Leino, and M. Moskal. The boogie verification debugger (tool paper). In *SEFM*, 2011.

[50] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to C. In *CAV*, pages 358–371, 2006.

[51] A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for C programs. *ENTCS*, 2007.

[52] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[53] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[54] J. V. Guttag and J. J. Horning. An introduction to the larch shared language. In *IFIP Congress*, pages 809–814, 1983.

[55] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. 10th SPIN Workshop on Software Model Checking*, 2003.

[56] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[57] T.-Y. Huang, P.-C. Chou, C.-H. Tsai, and H.-A. Chen. Automated fault localization with statistically suspicious program states. In *LCTES*, 2007.

[58] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2), Apr. 2002.

[59] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, 2006.

[60] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE*, pages 184–193, 2007.

[61] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE*, pages 184–193, 2007.

[62] T. Joachims. Making large-scale svm learning practical. *Advances in Kernel Methods - Support Vector Learning*, 1999.

[63] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *17th Conference on Computer Aided Verification (CAV '05)*, 2005.

[64] J. A. Jones. *Semi-Automatic Fault Localization*. PhD thesis, Georgia Institute of Technology, 2008.

[65] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *ISSTA*, pages 16–26, 2007.

[66] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.

[67] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. *CoRR*, abs/1011.1589, 2010.

[68] S. Khurshid, I. García, and Y. L. Suen. Repairing structurally complex data. In *12th SPIN Workshop on Model Checking of Software*, San Francisco, CA, Aug. 2005.

[69] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[70] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, 2010.

[71] V. Kuncak and M. C. Rinard. An overview of the jahob analysis system: project goals and current status. In *IPDPS*, 2006.

[72] T. Lev-Ami and S. Sagiv. Tvla: A system for implementing static analyses. In *SAS*, 2000.

[73] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.

[74] M. H. Liffiton and K. A. Sakallah. On finding all minimally unsatisfiable subformulas. In *SAT*, pages 173–186, 2005.

[75] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Software Eng.*, 32(10), 2006.

[76] F. Long and M. Rinard. Staged program repair with condition synthesis. In *FSE*, pages 166–178, 2015.

[77] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *ASE*, pages 615–619, Nov. 2009.

[78] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *ASE*, 2009.

[79] J. Marques-Silva and J. Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *DATE*, pages 408–413, 2008.

[80] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *ICSE*, pages 448–458, 2015.

[81] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[82] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[83] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535, 2001.

[84] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning report 02-3, May 2002.

[85] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 772–781, 2013.

[86] T. Parr and Others. Another tool for language recognition. `http://www.antlr.org/`.

[87] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September 8–10, 2003.

[88] R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1), 1986.

[89] X. Ren and B. G. Ryder. Heuristic ranking of java program edits for fault localization. In *ISSTA*, pages 239–249, 2007.

[90] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[91] S. Roychowdhury and S. Khurshid. A novel framework for locating software faults using latent divergences. In *ECML/PKDD (3)*, pages 49–64, 2011.

[92] S. Roychowdhury and S. Khurshid. Software fault localization using feature selection. In *MALETS*, pages 11–18, 2011.

[93] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, pages 56–66, 2009.

[94] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2005.

[95] F. Servant and J. A. Jones. Whosefault: Automatic developer-to-fault assignment through fault localization. In *ICSE*, pages 36–46, 2012.

[96] E. Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1982.

[97] R. Singh and A. Solar-Lezama. SPT: Storyboard programming tool. In *CAV*, pages 738–743, 2012.

[98] A. Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, pages 4–13, 2009.

[99] A. Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, pages 4–13, 2009.

[100] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[101] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM*, 2008.

[102] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Proc. 13th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Braga, Portugal, Mar. 2007.

[103] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag, 1995.

[104] M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. 9th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, April 2003.

[105] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[106] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA*, pages 61–72, 2010.

[107] W. Weimer. Patches as better bug reports. In *GPCE*, 2006.

[108] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[109] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[110] M. Weiser. Program slicing. In *Proc. 5th International Conference on Software Engineering (ICSE)*, San Diego, California, Mar. 1981. IEEE Computer Society Press.

[111] J. L. Wilkerson and D. Tauritz. Coevolutionary automated software correction. In *GECCO*, pages 1391–1392, 2010.

[112] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3), 2007.

[113] J. Zhang, S. Li, and S. Shen. Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In *AI*, 2006.

[114] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *ESEC/FSE*, 2009.