View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UT Digital Repository

Copyright
by
Kaiyuan Wang
2018

The Dissertation Committee for Kaiyuan Wang
certifies that this is the approved version of the following dissertation:

Automated Synthesis and Debugging of Declarative Models in Alloy

Committee:

Sarfraz Khurshid, Supervisor

Vijay Garg

Milos Gilgoric

Christine Julien

Darko Marinov

Automated Synthesis and Debugging of Declarative Models in Alloy

by

Kaiyuan Wang

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

August 2018

Dedicated to my family

Acknowledgments

First of all, I would like to thank my adviser, Sarfraz Khurshid. If it were not for him, I would have never finished my PhD. Sarfraz accepted me as a PhD student after I did a master thesis with him. He spent tremendous amount of time to discuss research ideas with me and helped me with different problems I encountered during my research. I will never forget the numerous nights we worked together to catch up with paper deadlines. I learned how to come up with research ideas and how to write technical papers from him and I feel very lucky to have him to be my adviser. Although I could never do enough to return all that he had done for me, I hope that I could advise students with the same passion some time in the future.

I would also like to thank Milos Gligoric for his rigorous training shortly after he joined University of Texas at Austin. During the time we worked together, I learned ways to set up experiments efficiently as well as writing good performance code. I really enjoy working with him. Although Milos is not my official co-adviser, he helped me a lot and I treat him as my co-adviser in my mind. I will miss the nights we worked together on our projects.

I want to thank Darko Marinov from University of Illinois at Urbana-Champaign for the nights we spent together to discuss and improve our projects. Darko is a very kind person who helped me a lot with my dissertation. He also

gave me advices about my future career and I really enjoy talking with him.

I would like to thank Vijay Garg, Milos Gligoric, Christine Julien and Darko Marinov for their generous help during my graduate studies. They served on my thesis committee and helped me improve the presentation of this dissertation.

None of the projects would have been fun and successful without my collaborators: Don Batory, Ahmet Celik, Milos Gligoric, Divya Gopinath, Jinru Hua, Sarfraz Khurshid, Jongwook Kim, Manos Koukoutos, Corina Pasareanu, Allison Sullivan, Zijiang Yang, Razieh Nokhbeh Zaeem, Mengshi Zhang, and Chenguang Zhu. I look forward to working with them again in the future.

I was lucky to have a number of office mates who share their thoughts and enrich my life at grad school: Ahmet Celik, Yen-Jung Chang, Hayes Converse, Nima Dini, Changyong Hu, Jinru Hua, Chenguang Liu, Alyas Mohammed, Pengyu Nie, Karl Palmskog, Rui Qiu, Shikhar Singh, Marko Vasic, Mengshi Zhang, Tianyi Zhang, Xiong Zheng, and Chenguang Zhu. I was also happy to work with several exceptional master students, including Oguz Demir, Jingjiang Li, Kewei Ma, Zijiang Yang, and Cagdas Yelen.

This work would not have been possible without the support from my family and friends. I especially thank my parent, Rong Xie and Zhen Wang, for their love, encouragement and trust.

Parts of this dissertation were published at ABZ 2018a [143] (Chapter 3); ABZ 2018b [146] and FSE Demo 2018 [144] (Chapter 4); and ASE

2018 [141] (Chapter 6). In addition, Chapter 5 is available on arXiv [145]. I would like to thank the anonymous reviewers and conference audience for their invaluable comments.

My research was funded in part by the National Science Foundation.

Automated Synthesis and Debugging of Declarative Models in Alloy

Publication No. _______________

Kaiyuan Wang, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Sarfraz Khurshid

In theory, formal specifications offer numerous benefits in developing more reliable software. In practice however, the use of specifications is rather limited, and practitioners often consider them more trouble than they are worth. Indeed, manually writing detailed specifications using notations that have unfamiliar syntax and semantics can be a daunting task – even for experienced programmers. We introduce a new automated approach for synthesis of desired specifications and debugging of faulty specifications using given examples that capture the essence of desired properties and serve as test cases. Our focus is specifications written in the declarative language Alloy – a first-order logic based on relations with transitive closure, and its SAT-based analysis engine. Our key insight is that a test-driven foundation enables modern approaches to synthesis and debugging of *imperative* code to serve as a basis for developing novel analogous techniques for *declarative* specifications. For synthesis, we build on equivalence in relational algebra and introduce techniques

for generating candidate Alloy expressions. We also introduce a technique to complete a partial Alloy model with holes using constraint solving. For locating faults in buggy specifications, we build on mutation-based fault localization and introduce techniques for locating likely faulty nodes in the abstract syntax tree of the faulty specification. Moreover, we integrate our expression generation and fault localization techniques to introduce a technique for automated specification repair. We experimentally evaluate our techniques using several Alloy models as subjects, including those with real faults. The results show that our techniques are effective at synthesis and debugging of the subjects. We believe our techniques provide an important step towards increasing the role of formal specifications in developing more reliable software and realizing their promise.

Table of Contents

Acknowledgments **v**

Abstract **viii**

List of Tables **xiii**

List of Figures **xiv**

Chapter 1. Introduction **1**
1.1 Thesis Overview 3
1.1.1 Expression generation for Alloy 5
1.1.2 Sketching for Alloy 6
1.1.3 Fault localization for Alloy 7
1.1.4 Automated repair for Alloy 9
1.2 Contributions 9
1.3 Organization 12

Chapter 2. Background **13**
2.1 Unit testing for Alloy 13
2.2 Mutation testing for Alloy 15

Chapter 3. Non-Equivalent Expression Generation[1] **17**
3.1 Example 17
3.2 RexGen Framework 20
3.2.1 Technique input 20
3.2.2 Generating expressions 21
3.3 Experimental evaluation 27
3.3.1 Evaluation models 27
3.3.2 RexGen results 29
3.4 Summary 32

Chapter 4. Solver-based Sketching of Alloy Models[2] **35**
4.1 Example 35
4.2 ASketch Framework 39
4.2.1 Input Language 39
4.2.2 Solver-based sketching 41
4.2.2.1 Parameterize Alloy constructs 42
4.2.2.2 Create Alloy meta constructs to encode holes . 43
4.2.2.3 Express test valuations as facts 46
4.2.2.4 Invoke Alloy Analyzer to complete holes 46
4.3 Evaluation 47
4.3.1 Sketching problems 47
4.3.2 ASketch results 48
4.4 Summary 52

Chapter 5. Fault Localization for Alloy **54**
5.1 Example 54
5.2 Technique 58
5.2.1 Suspiciousness Formulas 59
5.2.2 AlloyFL 60
5.3 Distance Metrics 66
5.4 Evaluation 69
5.4.1 Experiment Setting 70
5.4.2 RQ1: AlloyFL Accuracy and Time Overhead 72
5.4.3 RQ2: Suspiciousness Formula Impact 81
5.5 Summary 84

Chapter 6. Automated model repair for Alloy[3] **85**
6.1 Example 85
6.2 Technique 88
6.2.1 Create Holes 89
6.2.2 Generating Expressions 91
6.2.3 Search Strategies 93
6.2.4 Running Tests 96

6.2.5 Hierarchical Caching 98
6.2.6 Repair Algorithm 100
6.3 Evaluation 102
6.3.1 Experiment Setting 104
6.3.2 Repair Efficacy 105
6.3.3 Patch Quality 107
6.3.4 Limitation 109
6.4 Summary 110

Chapter 7. Related Work **112**
7.1 Expression Generation 112
7.2 Program Sketching 114
7.3 Fault Localization 115
7.4 Program Repair 117
7.5 Alloy 118

Chapter 8. Conclusion and Future Work **120**

Bibliography **122**

List of Tables

3.1 Static pruning rules 25
3.2 Syntactic approximation for $a \subseteq b$. $\cong$ means syntactic match. 26
3.3 Basic information of models used to evaluate RexGen 29
3.4 RexGen performance 34

4.1 Supported fragments for non-recursively defined holes 40
4.2 ASketch results for finding a solution. Times are in seconds. . 51

List of Figures

1.1 Techniques as a whole 4

2.1 Acyclic Singly Linked List. 14
2.2 Mutation Operators 16

4.1 Four test valuations 36

5.1 Faulty Farmer Example and MuAlloy Generated Tests. 55
5.2 Suspiciousness Formulas in AlloyFL. 59
5.3 Illustration of AlloyFL_{un} and AlloyFL_{su} 60
5.4 Distance Metrics Examples 67
5.5 Correct Models Information. 71
5.6 Distance Metrics for Real Faults. 73
5.7 Top-k Metrics for Real Faults. 76
5.8 Distance Metrics for Mutant Faults. 78
5.9 Top-k Metrics for Mutant Faults. 80
5.10 Formula Impact on AlloyFL for Real Faults. 82
5.11 Formulas Impact on AlloyFL for Mutant Faults. 82

6.1 First Suspicious Node for Faulty Farmer Example. 86
6.2 Patches for the faulty farmer model. 88
6.3 Hole creation schemas for Alloy Surface Syntax 90
6.4 Expression generation syntax. 92
6.5 All combinations partitions. 95
6.6 Dependency graph for `test1` in Figure 5.1b. 97
6.7 ARepair Results 103
6.8 Patches comparison 107

Chapter 1

Introduction

Building software models plays an important role in building reliable systems. Alloy is a well-known modeling language that has been used effectively in academic and industrial settings [28, 43, 88, 160]. Alloy models are declarative and consist of relational, first-order logic formulas with transitive closure. Two key strengths of Alloy are its expressive notation that allows succinctly writing complex structural properties, and its analyzability. Alloy has a SAT-based analyzer that performs automatic analysis within the user-defined *scope*, i.e., bound on the universe of discourse. Specifically, the analyzer finds *instances*, i.e., valuations for relations in the model such that the (chosen) formulas in the model evaluate to true. The analyzer can also find *counterexamples* that refute properties of interest; an instance for the negation of the property formula serves as a counterexample. Moreover, the analyzer can enumerate valuations and graphically display them.

While Alloy's expressive notation allows succinct formulation of complex properties, reasoning about the correctness of Alloy formulas, e.g., in the presence of quantification and transitive closure, requires much care. Because Alloy models are logical constraints, they can have two basic kinds of faults:

overconstraints that rule out valid valuations, and *underconstraints* that permit invalid valuations. Traditionally, Alloy users validate their models using two approaches: (1) use the analyzer to enumerate instances (and/or counterexamples) for their model and inspect them to see if some unexpected, invalid valuations are present and/or some expected, valid valuations are absent; or (2) write alternative constraints and use the analyzer to check for expected logical relationships, e.g., equivalence or implication between different formulas.

Consequently, correctly writing declarative models that represent non-trivial properties is not easy, especially for practitioners who are not well-versed with the intricate syntax and semantics of declarative languages. For example, in a declarative language, which lacks *sequencing* of operations, when trying to test, there is no notion of where the "execution" starts, how it proceeds, what conditional branches it encounters, what path it takes, and how the values of the program variables are updated and the final return value is computed. Indeed, even "what is the analog of a traditional test?" may not be formally defined.

Recent work introduced AUnit [131, 134], a unit testing framework for Alloy. AUnit defined test cases, test execution, and model coverage for Alloy. We propose four techniques on top of AUnit foundation. Specifically, our work leverages AUnit test cases. We first introduce a relational expression generator, called RexGen, which is able to generate non-equivalent expressions within a given bound. To evaluate RexGen, we use a set of well-studied Alloy

models and show that RexGen is able to prune a large number of equivalent expressions. We next introduce a sketching technique, called ASketch, which is able to complete a partial Alloy model with holes such that the completed model satisfies the properties captured by a give AUnit test suite. ASketch encodes a partial Alloy model, a set of given candidate fragments and an AUnit test suite into a meta Alloy model and invokes the SAT solver to search for solutions. To evaluate ASketch, we use a set of intricate Alloy formulas and show that ASketch is promising to sketch Alloy models. We next introduce a set of fault localization techniques, called AlloyFL, which takes as input a faulty Alloy model and a suite of AUnit tests. AlloyFL's output is a ranked list of Alloy AST nodes in the descending order of their suspiciousness. We evaluate AlloyFL using a set of real faults and injected faults. The experiment shows that AlloyFL is effective. Lastly, we introduce an automated repair tool, called ARepair, which takes as input a faulty Alloy model and a suite of AUnit tests. The output is a fixed model which makes all tests pass. We evaluated ARepair on real faulty Alloy models and the experimental results show that ARepair is effective.

1.1 Thesis Overview

This thesis introduces four techniques that work in tandem. Two of the techniques (RexGen and ASketch) address the core synthesis problem for Alloy, and two of them (AlloyFL and ARepair) address the core debugging problem of Alloy. All four of them apply in synergy to provide a viable solution to

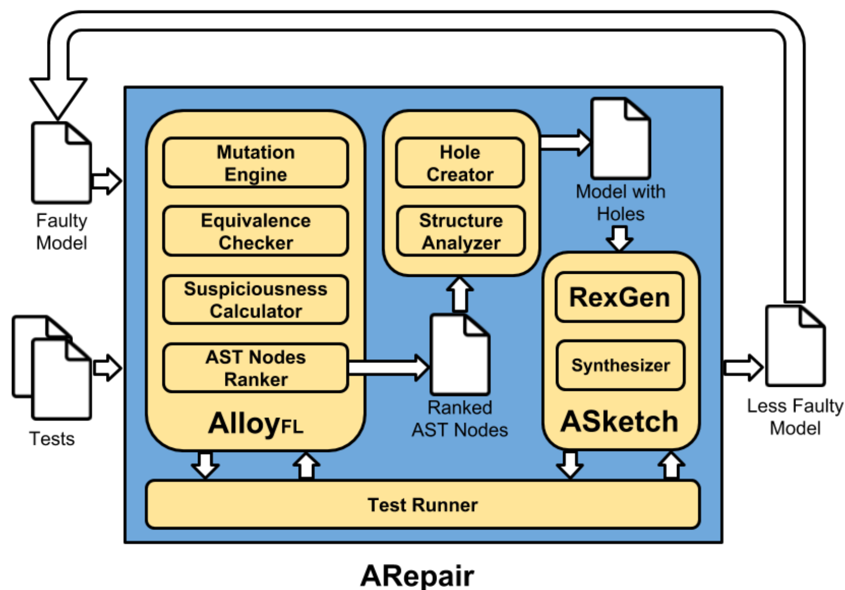

Figure 1.1: Techniques as a whole

synthesis-driven repair for faulty models (Figure 1.1). RexGen enriches the use of synthesis techniques for relational algebra. We show that it can be used together with ASketch to complete partial Alloy models. AlloyFL helps Alloy users to find fault locations and is shown to be more accurate than the Alloy built-in unsat core. We show that both RexGen and AlloyFL can be integrated as parts of ARepair which can automatically fix faulty Alloy models for users.

This thesis makes a four-fold contribution:

1. We introduce RexGen to reduce the search space of synthesis techniques proposed in this thesis.
2. We introduce ASketch to reduce the manual effort for writing completed Alloy model.
3. We introduce AlloyFL to reduce the manual effort for locating faults and debugging faulty Alloy models.
4. We introduce ARepair to reduce the manual effort for fixing faulty Alloy models.

1.1.1 Expression generation for Alloy

We introduce new techniques for generating non-equivalent relational expressions and embody the techniques in RexGen. RexGen takes as input a set of basic Alloy expressions and a bound on the target expression size, and outputs a set of Alloy expressions up to the specified size (built on top of the basic expressions). RexGen offers three automatic pruning modes for bottom-up generation of relational expressions. One mode, *static* pruning, directly prunes from generation many equivalent expressions based on a fixed suite of equivalence rules, which include well-known equivalences and also dozens more that we discovered using the Alloy analyzer. Another mode, *dynamic* pruning, uses the analyzer during generation to prune equivalent expressions incrementally by comparing each new expression to a representative from each equivalence class formed thus far, while forming new equivalence classes as needed. The third mode, *modulo-instance* pruning, allows the user to provide AUnit *test* valuations, and prunes an expression if it is equivalent to some

generated expression with respect to all given test valuations (even if not equivalent for some other valuations).

We perform an experimental evaluation of RexGen using expression generation problems derived from 12 Alloy models. We evaluate the number of expressions that RexGen generates and the time that RexGen takes to generate those expressions for each problem under different settings. The experimental results show that static pruning offers the best trade-off, creating mostly *semantically different* expressions, substantially reducing the number of expressions from simple grammar-based generation, while not increasing the generation time—in fact, often having smaller generation time than not using any equivalence pruning rules.

1.1.2 Sketching for Alloy

We introduce the first solver-based sketching technique for Alloy, called ASketch, which takes as input a partial Alloy model with holes, a candidate fragment generator and a suite of AUnit test cases that captures the desired model properties. The output is a complete Alloy model that makes all tests pass.

ASketch focuses on sketching several constructs of Alloy models, including relational expressions, logical operators, and quantifiers. Given a partial model and the corresponding test valuations, ASketch first parses the user-provided regular expressions and generates pools of matching fragments that can replace the holes. Then, ASketch systematically explores the resulting

search space of candidate Alloy models, to find a model that satisfies all test valuations. Specifically, ASketch uses constraint *solving* to explore the space of candidate models by creating one Alloy *meta-model* that encodes the model to sketch along with the fragments for holes and test valuations all at once. The meta-model effectively encodes multiple Alloy models, i.e., all models from the entire candidate space. Finally, ASketch uses the Alloy Analyzer to find solutions that can fill in the holes.

We perform an experimental evaluation of ASketch using 24 sketches derived from 5 core Alloy models. Experimental results show that ASketch can complete sketches with various holes and a large search space.

1.1.3 Fault localization for Alloy

We introduce the first fault localization technique for Alloy, called AlloyFL, which takes as input a faulty Alloy model and a suite of AUnit test cases that capture the desired model properties. The output is a list of Alloy AST nodes ranked in descending order of suspiciousness.

AlloyFL contains 5 techniques: AlloyFL$_{co}$, AlloyFL$_{un}$, AlloyFL$_{su}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$. AlloyFL$_{co}$ implements the *spectrum-based fault localization* (SBFL) technique [2, 41, 51, 95] for Alloy. AlloyFL$_{co}$ statically analyzes Alloy paragraphs that are transitively used in each test. Then, it ranks the Alloy paragraphs based on the number of passing/failing tests that invoke the paragraphs and a suspiciousness formula. AlloyFL$_{un}$ implements a SAT-based technique, which leverages the unsat core [120, 137, 138]. AlloyFL$_{un}$ collects all

AST nodes that are highlighted by the unsat core for each unsatisfiable failing test, and nodes highlighted more often are more likely to be faulty. AlloyFL$_{un}$ is designed to simulate how Alloy users would debug a faulty model manually using the unsat core. AlloyFL$_{su}$ is similar to AlloyFL$_{un}$ except that it uses both satisfiable and unsatisfiable failing tests to rank the AST nodes. AlloyFL$_{su}$ collects the nodes transitively used in satisfiable failing tests and nodes returned by unsat core in unsatisfiable failing tests. Nodes covered more often are ranked at the top. AlloyFL$_{mu}$ implements the *mutation-based fault localization* (MBFL) technique [92, 101] for Alloy. AlloyFL$_{mu}$ mutates Alloy AST nodes, e.g. "`a&&b`" to "`a||b`", to create non-equivalent mutants and check if the test results differ compared to the original model. AlloyFL$_{mu}$ uses a suspiciousness formula to compute the suspiciousness score for each mutant based on the number of passing/failing tests that kill the mutant. A test kills a mutant if its satisfiability changes compared to that of the original model. The node whose mutation gives the highest suspiciousness score, e.g. mutations on the node make almost all failing tests pass while preserving the results of passing tests, ranks at the top. The suspiciousness score of the mutated node conceptually propagate to all its descendants until mutations on its descendant nodes overwrite the corresponding suspiciousness scores. AlloyFL$_{hy}$ is a hybrid technique of both SBFL and MBFL. It takes the average of suspiciousness scores obtained from both AlloyFL$_{co}$ and AlloyFL$_{mu}$ and assign the score to the corresponding AST node. If an AST node is not mutable and does not have a suspiciousness score from AlloyFL$_{mu}$, then the suspiciousness score

from AlloyFL$_{co}$ is used. Finally, if multiple nodes have the same suspiciousness score, then AlloyFL prioritizes the nodes with less descendants.

We evaluate AlloyFL on a set of real world faults as well as injected faults. The experimental results on 38 real faults and 9000 mutant faults show that AlloyFL$_{mu}$ and AlloyFL$_{hy}$ are significantly more accurate than the baseline techniques, i.e. AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$.

1.1.4 Automated repair for Alloy

We introduce the first automated repair technique for Alloy, called ARepair, which takes as input a faulty Alloy model and a suite of AUnit tests that reveal the faults. The output is either a fixed model that pass all tests or a partially fixed model if ARepair cannot fix it.

1.2 Contributions

We make the following contributions in this dissertation.

RexGen

- **Problem.** We are the first to study the problem of expression generation for relational algebra.
- **Optimizations.** We introduce a suite of equivalence pruning rules for relational expressions to improve the efficacy of expression generation. We also build techniques on top of these equivalence pruning rules and

show that these techniques can significantly reduce the number of generated expressions.

- **Experiments.** We present an experimental evaluation based on problems derived from a set of Alloy models; the results show that RexGen with static pruning offers a promising approach for generating non-equivalent relational expressions.

ASketch

- **Idea.** We introduce the idea of sketching Alloy models using AUnit test valuations.

- **Technique.** We introduce a technique for completing Alloy sketches based on constraint solving.

- **Experiments.** We present an experimental evaluation with intricate Alloy formulas; the results show that ASketch introduces a promising approach for sketching Alloy models.

AlloyFL

- **Technique.** We propose, AlloyFL, the first set of AST node level fault localization techniques for Alloy that leverage multiple tests.

- **Metrics.** We follow the spirit of an existing nearest neighbor distance metric [111] and define 3 new distance metrics at the AST level to measure the accuracy of AlloyFL.

- **Evaluation.** We evaluate AlloyFL using 38 real faults and 9000 mutant faults derived from 18 existing models. The subject models all contain 1 or more faults and our experimental results show that MBFL techniques are significantly more accurate than the baseline SBFL techniques and SAT-based techniques.

ARepair

- **Alloy Model Repair.** ARepair is the first repair technique for Alloy, which uses both mutations and synthesis to repair faulty models. The experimental results show that the combined approach works well and many faulty models require both mutations and synthesis for a complete fix. ARepair does not require isolated faults. It can fix models with multiple faults or faults involving multiple locations.

- **Optimizations for Practical Model Repair.** ARepair does not search for fault patterns and apply repair templates to fix faults. Instead, it tries to repair a faulty AST in a bottom-up fashion, so it is more likely to repair faults with unseen patterns. The absence of repair templates results in an immense search space and we implement the following optimizations to make the technique tractable and reduce end-to-end time. (1) The expression generator RexGen prunes equivalent expressions based on equivalence pruning rules and modulo test inputs [68]. (2) The enumeration based approach explores the search space without expensive constraint solving [8]. (3) The construction of

dependency graphs for constraint formulas enables a small number of evaluator calls during the enumeration based approach. (4) The base-choice search strategy reduces the exploration space. (5) The hierarchical caching reduces sizes of the inputs to evaluator calls.

- **Evaluation.** We evaluate ARepair on real faults and the experimental results show its efficacy. We qualitatively compare patches generated by ARepair and human written patches, and show that the quality of the generated patches is good.

1.3 Organization

The rest of the thesis is organized as follows:

Chapter 2 describes the background of AUnit and MuAlloy. AUnit tests serve as the foundation of all techniques in this thesis. MuAlloy is a mutation testing framework for Alloy and AlloyFL is partly built on top of it. MuAlloy can automatically generate tests that kill mutated Alloy models and these tests are heavily used to evaluate AlloyFL and ARepair. Chapter 3 describes the expression generation technique RexGen. Chapter 4 describes the sketching technique ASketch. Chapter 5 described the fault localization technique AlloyFL. Chapter 6 describes the automated repair technique ARepair. Chapter 7 describes the related work. Finally, Chapter 8 concludes the thesis and discuss about the future work.

Chapter 2

Background

2.1 Unit testing for Alloy

Conceptually, a test case in AUnit is a pair that consists of an Alloy valuation and a command. To illustrate, Figure 2.1 (incorrectly) models an *acyclic* singly-linked list. The *signature* (`sig`) declaration "`one sig Node`" introduces a singleton set of list atom; "`sig Node`" introduces a set of node atoms. Each signature declaration also introduces a binary relation. The relation `header` maps `List` atoms to `Node` atoms. The relation `link` maps `Node` to `Node`. The keyword `lone` constrains the corresponding relation to be a partial function; therefore, each of `header` and `link` is a partial function. The predicate body contains an implication (`=>`) and intends to state that if the list has some header node, there exists a node reachable from the header with no link, i.e., the list is "null-terminated". The formula "`some List.header`" declares that the list has a header. The operator "`.`" is relational composition and "`^`" is transitive closure. The expression "`l.header.^link`" represents the set of nodes reachable from `l`'s header following one or more traversals along `link`. Structuring `Acyclic` as an implication allows for the list to be empty, as without the implication, the existential quantified formula requires there to be at least one `Node` atom in the `List`. The test predicate `testNoHeader` states that

```
one sig List { header: lone Node }
sig Node { link: set Node }
pred Acyclic() {
  some List.header => some n: List.header.^link | no n.link }

pred testNoHeader() {
  some disj L: List {
    List = L
    no Node
    no header
    no link
    Acyclic[] }}
run testNoHeader expect 1

pred testOneHeader() {
  some disj L: List {
    some disj N: Node {
      List = L
      Node = N
      header = L -> N
      no link
      Acyclic[] }}}
run testOneHeader expect 1
```

Figure 2.1: Acyclic Singly Linked List.

`List` has one atom and no `Node`, `header` or `link` is allowed. The corresponding run command enforces the valuation in `testNoHeader` and it is expected to be satisfiable. Another test predicate `testOneHeader` states that `List` and `Node` are singleton sets; `header` maps the `List` atom to the `Node` atom; and no `link` is allowed. The corresponding run command enforces the valuation in `testOneHeader` and it is expected to be satisfiable.

In this case, `testNoHeader` is satisfiable and it passes, but `testOneHeader`

is unsatisfiable and it fails. The fault is in the use of *transitive closure* `"^"` instead of *reflexive transitive closure* `"*"`.

2.2 Mutation testing for Alloy

Mutation testing is a powerful methodology for evaluating test suite quality [49]. AlloyFL borrows some of the mutation operators defined in previous work (MuAlloy [133, 140, 142]). Our mutation testing approach MuAlloy brings the spirit of traditional mutation testing to Alloy and defines how to create mutants of Alloy models, compute mutation testing results, and check for equivalent mutants. We introduce a suite of mutation operators for Alloy and introduce the use of SAT-based analysis for mutation testing and equivalent mutant checking. We transform the equivalent mutant checking problem into a satisfiability check such that if there is a solution to the resulting formula, we get a test that kills the mutant. While Alloy is based on first-order logic, checking for equivalent mutants can lead to higher-order solving, which the standard Alloy analyzer does not support. We ignore higher-order equivalence checking in this thesis for simplicity.

Figure 2.2 shows the mutation operators supported in MuAlloy. *MOR* mutates signature multiplicity, e.g. `"lone sig"` to `"one sig"`. *QOR* mutates quantifiers, e.g. `all` to `some`. *UOR*, *BOR* and *LOR* define operator replacement for unary, binary and formula list operators, respectively. For example, *UOR* mutates `a.*b` to `a.^b`; *BOR* mutates `a=>b` to `a<=>b`; and *LOR* mutates `a&&b` to `a||b`. *UOI* inserts an unary operator before expressions, e.g. `a.b` to `a.~b`.

Mutation Operator	**Description**
MOR	Multiplicity Operator Replacement
QOR	Quantifier Operator Replacement
UOR	Unary Operator Replacement
BOR	Binary Operator Replacement
LOR	Formula List Operator Replacement
UOI	Unary Operator Insertion
UOD	Unary Operator Deletion
LOD	Logical Operand Deletion
PBD	Paragraph Body Deletion
BOE	Binary Operand Exchange
IEOE	Imply-Else Operand Exchange

Figure 2.2: Mutation Operators

UOD deletes an unary operator, e.g. `a.* ~b` to `a.*b`. *LOD* deletes an operand of a logical operator, e.g. `a||b` to `b`. *PBD* deletes the body of an Alloy paragraph. *BOE* exchanges operands for a binary operator, e.g. `a=>b` to `b=>a`. *IEOE* exchanges the operands of `imply-else` operation, e.g. "`a => b else c`" to "`a => c else b`". All mutation operators are defined at the AST level and modifying AST nodes properly is non-trivial. For example, `&&` and `||` are list operators in Alloy. Replacing `&&` with `||` in "`a || (b && (c || d))`" should results in "`a || b || c || d`", which means we need to properly flatten the parent and child AST nodes after mutation.

Chapter 3

Non-Equivalent Expression Generation[1]

In this chapter, we introduce RexGen – the first generator for semantically non-equivalent relational expressions. We present the algorithms that define our generator, its embodiment based on the Alloy tool-set, and an experimental evaluation to show the effectiveness of its non-equivalent generation for a variety of problems with relational constraints.

The rest of the chapter is organized as follows. Section 3.1 presents an example. Section 3.2 describes in detail the techniques to generate non-equivalent relational expressions. Section 3.3 evaluates RexGen. Section 3.4 summarizes the chapter.

3.1 Example

We next present an example model to motivate relational expression generation and introduce the basic concepts of our approach. Consider this small but illustrative Alloy model of directed trees, adapted from a recent paper [96]:

[1] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. Systematic generation of non-equivalent expressions for relational algebra. In ABZ, 2018.

```
sig Node { edges: set Node }
pred Acyclic { no iden & ^edges }
pred Injective { edges.~edges in iden }
pred Connected { (Node -> Node) in ^(edges + ~edges) }
pred isDirectedTree { Acyclic and Injective and Connected }
run isDirectedTree for 4 Node
```

The model declares a set (called *signature* in Alloy) of nodes with a *field* called `edges` that is a binary relation of type `Node`×`Node`. The keyword `set` declares an arbitrary relation; Alloy also has keywords `one` and `lone` to constrain the relation to be a total or partial function, respectively. The *predicate* (`pred`) is a named formula that can be invoked elsewhere. The conjunction of `Acyclic`, `Injective`, and `Connected` would precisely represent directed trees. The binary operator `&` is set intersection; `+` is set union; "`in`" is subset; "`.`" is relational join; and `->` is Cartesian product. The (prefix) unary operator `^` is transitive closure, and ∼ is transpose; Alloy also has *reflexive* transitive closure (∗). The keyword `iden` represents the identity relation. The formula "`no E`" for expression `E` constrains `E` to be the empty set. The `run` command runs a given formula, and presents an instance of the given formula if the formula is satisfiable. The scope of 4 instructs the analyzer to create an instance with at most 4 nodes.

To illustrate expression generation using our approach, consider the signature declaration in this model, which introduces one set (`Node`) and one binary relation (`edges`). Given those declarations, a user may want to generate various expressions, e.g., in synthesis or repair tasks. For example, many Alloy beginners write "`pred Acyclic' { all n: Node | n !in n.^edges }`" and

may want to know if there is a semantically equivalent formula without any quantified variables (as in `Acyclic`). In that case, the user may want to systematically try `{UO E}` where `UO` represents any unary operator (`no`, `some`, `lone`, `one`) and `E` represents any valid expression, such that the formula `{UO E}` is equivalent to `Acyclic'`.

Assume we set the maximum size of any generated expression to 5, which suffices to generate even the largest relational expressions in this particular model. RexGen generates 581 expressions with no pruning, 438 with AC pruning (i.e., associativity and commutativity), 116 with static pruning, 105 with dynamic pruning, and 102 with modulo-instance pruning (for 14 tests). The generation time is largest for dynamic pruning, which uses Alloy analyzer to check each equivalence and takes 2.8 sec; in all other cases, no constraint solving is used, and the generation time is <1 sec. The following shows some of the equivalences discovered with dynamic pruning (where `univ` denotes the universe of discourse, which is equal to `Node` in the example model):

```
Node->Node = univ->univ               | (~edges)&(^edges) = (~edges)&(*edges)
(~edges).Node = Node.edges            | *((^edges)-edges) = *((*edges)-edges)
edges.(Node.edges) = edges.Node       | ^(edges.(^edges)) = edges.(^edges)
```

To illustrate generation of larger expressions, consider size 7. RexGen generates 17109 expressions with no pruning, 11191 with AC pruning, 1464 with static pruning, 771 with dynamic pruning, and 691 with modulo-instance pruning (for 14 tests). The generation time for dynamic pruning increases to 82.3 sec, for modulo-instance pruning increases to 1.7 sec, and for

the other techniques remains $<$1 sec. Thus, for this example, static pruning reduces the number of expressions by 86.9% over AC pruning while taking a similar amount of time; dynamic pruning reduces the number by 47.3% over static pruning but takes much longer due to many SAT calls. Moreover, modulo-instance pruning creates a similar number of expressions as dynamic pruning, which indicates the diversity of the tests, but takes less time due to not making SAT calls.

3.2 RexGen Framework

We next present our ***Re****lational* ***Ex****pression* ***Gen****erator* (RexGen) approach for generating non-equivalent relational expressions. We first describe the technique input and then the expression generation techniques.

3.2.1 Technique input

RexGen takes as input (1) a number of sets (signatures), relations (fields), and variables declared in an Alloy model (in the context in which the expressions should be generated), (2) a limit on the size of generated expressions, (3) optionally a target arity of expressions to generate, and (4) optionally a number of test valuations, i.e., values for the input sets and relations (but not for the bound variables). RexGen generates expressions using the following

grammar:

$$\begin{aligned} expr &::= expr\ binOp\ expr \ \mid\ expr^{*} \ \mid\ expr^{+} \ \mid\ expr^{-1} \ \mid\ terminal \\ binOp &::= \cup \ \mid\ \cap \ \mid\ \setminus \ \mid\ \times \ \mid\ \bowtie \\ terminal &::= set \ \mid\ relation \ \mid\ variable \end{aligned}$$

The grammar captures a subset of syntactically possible Alloy expressions, which cover a large space of candidate expressions likely to be intended by Alloy users. For example, we do not consider rarely used Alloy operators such as domain restriction (`<:`). We use standard notation of relational algebra: $\cup$ is set union, $\cap$ is set intersection, $\setminus$ is set difference, $\times$ is Cartesian product, $\bowtie$ is the relational join; e^*, e^+, e^{-1} denote the reflexive transitive closure, transitive closure, and transpose of e, respectively. Additionally we use the empty set $\varnothing$, the universal set $univ$, and the identity $iden = \{(x, x)|x \in univ\}$.

To systematically generate expressions, RexGen limits: (1) the size of expressions and (2) the maximum arity of expressions. There are different ways to define expression size; we consider the number of AST nodes in the expression: $size(terminal) = 1, size(e_1\ binOp\ e_2) = size(e_1) + size(e_2) + 1, size(expr^{unOp}) = size(expr) + 1$.

3.2.2 Generating expressions

We next describe how RexGen enumerates expressions within the given limits. In the spirit of synthesis tools [8], enumeration works bottom-up, starting from *terminal* expressions (sets, relations, and variables given as inputs)

and then iteratively combining smaller expressions to generate larger ones.

Our key contribution is pruning that aggressively removes expressions to increase the efficiency of the generation and/or reduce the number of generated expressions. The goal of pruning is to eliminate expressions that are semantically equivalent with previously generated expressions. Pruning has three modes: *static*, *dynamic*, and *modulo* pruning.

Expression generation algorithm. The generation algorithm maintains a list of expressions, $exprs[arity]$, indexed by the arity. The list maintains a total order among expressions of the same arity; we use $ind(e)$ to denote the index of the expression e in the list, and some pruning rules use this index.

The lists are instantiated with the terminal expressions (i.e., sets, relations, and variables declared in the model), based on their arity. The size of these expressions is 1. Then, until a limit is reached, the algorithm iteratively increases size and combines every operator and every combination of expressions of appropriate smaller sizes to generate expressions of the larger size. Each generated expression is then added to $exprs$ if it is (1) within the limits given for the generation, (2) well typed in Alloy, and (3) not pruned by the current pruning mode. Note that, by construction, expressions in $exprs$ are syntactically different. The rest of this section explains in detail well typedness and the three pruning modes.

Well typedness. RexGen tracks type information for generated expressions, typically using the default Alloy type system, which includes subset/subtyping

and union types [42]. However, for some expressions, RexGen tracks a more precise type than the default type system. The main reason is the semantics of reflexive transitive closure (*). In Alloy, reflexive transitive closure is a superset of the identity relation for the union of all sets (*univ*) and thus has type $univ \times univ$. For example, if a model has two sets, $Node$ and $Value$, and a relation, $edges$, of type $Node \times Node$, then $edges^*$ is not of type $Node \times Node$ but $univ \times univ$, where $univ = Node \cup Value$. However, this type is too broad; it allows for arbitrary applications of other operators and makes expression generation intractable, producing expressions that are not intended in practical use.

For example, consider the expression $a^* + b$, where a has type $A \times A$. Intuitively, we want to allow only expressions of type $A \times A$ for b; however, we cannot track this precisely if we allow a^* to have type $univ \times univ$. On the other hand, we cannot consider a^* to have type $A \times A$ because that would make a^* a subset of $A \times A$, causing the static pruning to incorrectly prune expressions like $a^* + A \times A$. Therefore, RexGen conceptually uses a special type system to type intermediate generated expressions, but uses Alloy type for static pruning.

Static pruning. Static pruning removes expressions that are known to be semantically equivalent with other generated expressions. This pruning considers equivalence with respect to *all* possible valuations not only given test valuations. To prune equivalent expressions, we derive a comprehensive suite of equivalence rules specific to relational algebra. Other generation systems [106]

use similar pruning rules for other domains, but our work is the first to provide rules specific to relational algebra.

Table 3.1 presents the static pruning rules of RexGen. The first column gives the pattern of equivalent expressions that the rule intends to eliminate. RexGen prunes the expression whose syntactic shape is the left-hand side of the equivalence. The second column specifies the condition for pruning. Note that almost all rules use only syntactic information or type (and arity) information for the involved expressions, which makes the rules easily checkable. An exception are a few rules that check the subset property between two sets/relations; because subset is a semantic property and not easily checkable, we approximate it conservatively, as shown in Table 3.2. Another exception is the rule for commutativity. To avoid generating both $a\ op\ b$ and $b\ op\ a$, where op is a commutative operation, we use the total order defined for each arity by $exprs$: we prune the expression with $ind(a) > ind(b)$, where $ind(e)$ is the index of e in the list $exprs$.

Dynamic pruning. Dynamic pruning removes equivalent expressions by using the Alloy analyzer to check whether an expression is equivalent to another one already generated. Unlike static pruning, dynamic pruning considers (1) all signature/field constraints (e.g., that a relation must be a function) and (2) bound variables in the scope of the generated expression. To our knowledge, no previous work handles variables locally bound by a quantifier in the scope of the generated expression.

For a new expression, E, and a previously generated expression, E',

Table 3.1: Static pruning rules

Equivalence (lhs = rhs)	Condition if needed; otherwise `true`
$a\ op\ (b\ op\ c) = (a\ op\ b)\ op\ c$	op associative
$a\ op\ b = b\ op\ a$	op commutative and $ind(a) > ind(b)$
$a \cup b = b$ and $b \cup a = b$	$[\![a]\!] \subseteq [\![b]\!]$
– Similar for $\cap$ and $\supseteq$	
$a \setminus b = \varnothing$	$[\![a]\!] \subseteq [\![b]\!]$
$a \cup b = c \cup b$	$\exists c.a \cong c \cup b$ or $a \cong b \cup c$ or $a \cong c \setminus b$
$a \cup b = b$	$\exists c.a \cong c \cap b$ or $a \cong b \cap c$ or $a \cong b \setminus c$
– Also symmetrically	– where $\cong$ is syntactic pattern matching
$(a\ op_1\ b)\ op_2\ (a\ op_1\ c) = a\ op_1\ (b\ op_2\ c)$	$op_1 \in \{\bowtie, \times, \cap\}, op_2 \in \{\cup, \cap\}$
– Similar for $(a\ op_1\ b)\ op_2\ (c\ op_1\ b)$	
$a^{-1}\ op\ b^{-1} = (a\ op\ b)^{-1}$	$op \in \{\cup, \cap, \setminus, \bowtie\}$
$\bigcup e_i = \bigcup_{i \neq j} e_i$,	$e_j \cong e_k$ for some $j \neq k$
– Similar for $\bigcap$	
$a \setminus (b \cup c) = (a \setminus b) \setminus c$	
$a \setminus (a \cap b) = a \setminus b$	
– Similar for $a \setminus (b \cap a)$	
$a \setminus (a \setminus b) = a \cap b$	
$a \setminus (b \setminus a) = a$	
$(a \cup b) \setminus a = b \setminus a$	
$(a\ op\ b) \setminus (a\ op\ c) = a\ op\ (b \setminus c)$	$op \in \{\times, \cap\}$
$(a \cap b) \setminus c = a \cap (b \setminus c)$	
$a \bowtie (a \times b) = b$	$card(a) \geq 1$
– Similar for $(b \times a) \bowtie a$	
$a \bowtie b^{-1} = b \bowtie a$	$arity(a) = 1$
$A \bowtie b^* = A$	$b : A \times A$
– Similar for $b^* \bowtie A$	– where $b : A \times A$ means that b has type $A \times A$
$A \bowtie b^+ = A \bowtie b$	$b : A \times A$
$b^+ \bowtie A = b \bowtie A$	$b : A \times A$
$b \bowtie b^* = b^+$	
– Similar for $b^* \bowtie b$	
$a^{*+} = a^*$	
– Similar for a^{+*}	
$a^{-1^{-1}} = a$	
$a^{*-1} = a^{-1*}$	
$a^{+-1} = a^{-1+}$	
$(a\ op\ b^{-1})^{-1} = a^{-1}\ op\ b$	$op \in \{\cup, \cap, \setminus, \bowtie\}$
$(a \times b)^+ = a \times b$	
$a \bowtie (b \times c) = (a \bowtie b) \times c$	$arity(a) + arity(b) > 2$
– Similar for $(a \times b) \bowtie c$	
$b^{-1} \bowtie a = a \bowtie b$	$arity(a) = 1$
$a^+ \bowtie a = a \bowtie a^+$	
$a^* \bowtie a^* = a^*$	
$a^* \bowtie a^+ = a^+$	
– Similar for $a^+ \bowtie a^*$	
$a^+ \bowtie a^+ = a \bowtie a^+$	
$(a \setminus b) \bowtie (b \times c) = \varnothing$	
– Also symmetrically	
$a \bowtie ((b \setminus a) \times c) = \varnothing$	
– Also symmetrically	
$A \bowtie (A \times b) = b$	$arity(A) = 1$
– Similar for $(b \times A) \bowtie A$	$b : B_1 \times ... \times A \times ... \times B_n$ for some $B_i = A$

Table 3.2: Syntactic approximation for $a \subseteq b$. $\cong$ means syntactic match.

1. $b \cong A$, $a : A$	5. $a \cong b \setminus c$
2. $a \cong b$	6. $a \cong c^+, b \cong c^*$
3. $b \cong a \cup c$ or $b \cong c \cup a$	7. $a \cong c \bowtie c \bowtie \ldots \bowtie c, b \cong c^+$ or $b \cong c^*$
4. $a \cong b \cap c$ or $a \cong c \cap b$	8. $a \cong c \times c$, $b \cong d^*$, a has cardinality 1, c has arity 1

RexGen creates a new Alloy model that includes all sig/field declarations from the RexGen input plus `check { all` v_1`:` D_1 `|...| all` v_n`:` D_n `|` E `=` E' `}`, where $v_1 \ldots v_n$ are variables used in the two expressions (except for sigs/fields from the model) and $D_1 \ldots D_n$ are their corresponding domains. For example, if E is `n.~edges` and E' is `Node.*edges`, then the equivalence checking command is `check { all n: Node | n.~edges = Node.*edges }`. This check is issued for every previously generated expression in *exprs* until either the new expression is found equivalent to some previously generated one, or the new expression is found not equivalent to any previously generated one and is thus added to *exprs*. Dynamic pruning can be applied to all expressions for every arity or only expressions of the target arity.

Modulo pruning. Modulo pruning [68] removes equivalent expressions based on their values for the user-given valuations of the input test suite. Specifically, modulo pruning builds equivalence classes of expressions by grouping together all expressions that *evaluate* to the same value across all test valuations, and keeping only one expression per equivalence class.

Modulo pruning determines an expression's equivalence class without constraint *solving*, by utilizing the `Evaluator` feature of the Alloy Analyzer to

perform constraint *checking*. The `Evaluator` takes as input an Alloy instance and an Alloy expression, and returns the concrete value of the expression for the given instance. For a new expression E, modulo pruning evaluates E for every test valuation in the suite, building a map of E's concrete values. If E contains any free variable(s), modulo pruning evaluates E for each element in the variable's domain, or more generally, for the cross product of domain elements if E contains multiple variables. If E's concrete-value map matches a previous expression, then E is pruned out; otherwise, E is kept. Modulo pruning determines equivalence based on the user-given test suite, not guaranteeing equivalence across all instances in scope as dynamic pruning does.

3.3 Experimental evaluation

We next present our experimental evaluation of RexGen. We use 12 diverse Alloy models for evaluation (Section 3.3.1). We evaluate the number of expressions RexGen generates and the time it takes for each model under different settings (Section 3.3.2).

3.3.1 Evaluation models

We evaluate RexGen using 12 models comprised of a wide variety of example, educational, and "real-world" specifications. Address book (**addr**), Dijkstra mutual exclusion algorithm (**dijkstra**), farmer crossing-river puzzle (**farmer**), Halmos handshake problem (**hshake**), and genealogy (**gene**) are from the Alloy's distribution examples. Bad employee (**bempl**), colored tree

(**ctree**), directed tree (**dtree**), and grade book (**grade**) are Alloy translations of access-control specifications used to evaluate existing scenario-finding work [96, 115]. Binary tree (**btree**) constrains the graph to be a binary tree. Propositional resolution (**resfm**) is from Torlak et al. [137]. Singly linked list (**sll**) models acyclic lists.

Table 3.3 shows the basic information of these models. **Model** is the name. #**AST** is the number of AST nodes in each model. #**Sig** is the number of signatures declared in each model. #**Rel** is the number of relations declared in each model. For each model, we find all identifiers in scope, including signatures, relations, and bound variables, for the *largest* expression (w.r.t. our measure of size). #**Var** is the number of all identifiers in scope to generate expressions. In our experiment, we first find the expression with the largest size in each model and then use all sigs, relations, and variables in the scope of that expression to generate more expressions. #**PrimVar** is the number of primary variables when we run an empty command (`run {}`) without test-specific constraints; it represents the basic complexity of signature declarations and constraints that always hold in each model. #**Test** is the number of tests; we use the same number of tests for each model so that the results do not depend on the number of tests. We chose the number of tests based on the *sll* model, where we create tests such that modulo pruning generates the same number of expressions of size 4 as dynamic pruning for this model. We iteratively add tests until modulo pruning and dynamic pruning create the same set of expressions. In the end, we obtain 14 tests for **sll** and use the

Table 3.3: Basic information of models used to evaluate RexGen

Model	**#AST**	**#Sig**	**#Rel**	**#Var**	**#PrimVar**	**#Test**
addr	114	4	2	8	45	14
bempl	46	6	3	11	38	14
btree	53	2	2	6	24	14
ctree	71	4	2	8	18	14
dijkstra	385	3	1	10	57	14
dtree	49	1	1	2	12	14
farmer	169	6	3	14	24	14
gene	139	5	2	8	20	14
grade	64	5	4	11	48	14
hshake	127	3	2	6	19	14
resfm	285	8	7	19	101	14
sll	33	2	2	5	15	14

same number of tests for other models.

Our experiments are performed on a MacBook Pro running OS X El Capitan with 2.5 GHz Intel Core i7-4870HQ and 16GB of RAM.

3.3.2 RexGen results

Table 3.4 shows the performance of RexGen across different expression pruning environments: **No Pr.** uses no pruning rules, **AC Pr.** uses just associativity and commutativity pruning rules, **Static Pr.** uses all static pruning rules, **Dynamic Pr.** uses dynamic pruning, and **Modulo Pr.** uses modulo-instance pruning. Note that both dynamic pruning and modulo-instance pruning are applied on expressions after they are pruned by static pruning. Column **Problem** shows the Alloy model and the corresponding size used for generation. For each pruning environment, #**expr** shows the number of expressions

generated and **time** shows the time duration in milliseconds to generate all expressions, with a time-out of one hour. The number of generated expressions shown in the table is for expressions of all arities up to 3.

Expression generation using **No Pr.**, **AC Pr.**, and **Static Pr.** is fast, taking at most 7.9 seconds (**farmer** and size 7 using **No Pr.**), but frequently finishing in under a second. Accordingly, both **AC Pr.** and **Static Pr.** have negligible overhead. However, the number of expressions generated can vary greatly, as seen in Table 3.4. **No Pr.** generates all possible expressions and provides a means of measuring the effectiveness of different pruning environments. Compared to **No Pr.**, **AC Pr.** reduces the number of expressions generated by 8.7–54.7%, while **Static Pr.** reduces the number of expressions generated by 36.6–91.4%. Compared directly, **Static Pr.** generates 28.4–86.9% fewer expressions than **AC Pr.**. In other words, **Static Pr.**'s additional pruning rules highlight that associativity and commutativity are not strong enough to prune relational expressions on their own. Moreover, Table 3.4 shows that the pruning rules for **AC Pr.** and **Static Pr.** reduce the space of possible expressions by a large enough degree that both techniques often finish faster than **No Pr.**, despite the time they spend on applying equivalence rules to check expressions. Although **Static Pr.** has 40 more rules than **AC Pr.**, the difference in runtime between **AC Pr.** and **Static Pr.** is often less than a second. Therefore, **Static Pr.**'s rules are inexpensive to run but effective at reducing the number of generated expressions.

We can analyze expressions to prune out more equivalences. **Dynamic**

Pr. further prunes expressions generated by **Static Pr.**; **Dynamic Pr.** is motivated by using Alloy to find all equivalences (within a given scope), thus capturing equivalences which cannot be captured by generic static pruning rules. As expected, **Dynamic Pr.** reduces the number of expressions from **Static Pr.**, by 3.6–71.4%. **Dynamic Pr.** gives the minimum number of non-equivalent expressions for each model, showing the lower bound of what **Static Pr.** could achieve.

Modulo Pr. also filters expressions generated by **Static Pr.**; specifically, **Modulo Pr.** reduces the expressions from **Static Pr.** by 5.0–91.3%. **Dynamic Pr.** can be viewed as **Modulo Pr.** if the input test suite covered all instances in scope. However, since we use only 14 tests per model for **Modulo Pr.**, **Modulo Pr.** may even prune expressions that are semantically non-equivalent up to a given scope but equivalent over all 14 tests. For example, **Modulo Pr.** prunes 10 more size 4 expressions and 223 more size 5 expressions for **addr** compared to **Dynamic Pr.**. Therefore, as expected, **Modulo Pr.** can reduce the number of generated expressions compared to **Dynamic Pr.**, by as much as 50.2% (**addr**), or **Modulo Pr.** can generate the same number of expressions (**sll** and size 4) but 5.2× faster. The trade-off is that, while **Dynamic Pr.** is guaranteed to not prune expressions that are semantically non-equivalent within a given scope, it is slower than **Modulo Pr.**; **Dynamic Pr.** times out on 7 different problems, while **Modulo Pr.** frequently finishes in under a minute, with the longest runtime being 2156.7 seconds. While **Modulo Pr.** provides a practical, lighter-weight alternative

to **Dynamic Pr.**, **Modulo Pr.** still has a high overhead over **Static Pr.**. For instance, for **farmer** and size 7, **Static Pr.** can generate expressions in 4.0 seconds, while **Modulo Pr.** needs 2156.7 seconds to finish.

In our experiment, applying **Dynamic Pr.** or **Modulo Pr.** on expressions generated with **No Pr.** or **AC Pr.** takes significantly longer. **Static Pr.**'s ability to significantly reduce the number of generated expressions, with a negligible overhead, makes **Static Pr.** the recommended approach for relational expression generation (even when considering more advanced pruning techniques like **Dynamic Pr.** or **Modulo Pr.**). To check that our static pruning rules are correct, we ran dynamic pruning on expressions generated using **AC Pr.** and **Static Pr.**: we found that the numbers of non-equivalent expressions generated after dynamic pruning for both **AC Pr.** and **Static Pr.** are exactly the same, which indicates that **Static Pr.** does not incorrectly prune any non-equivalent expression.

3.4 Summary

We introduced RexGen, the first generator for non-equivalent relational expressions. We presented a set of equivalence rules for relational expressions, used them for pruning in our generator, embodied the generator based on the Alloy tool-set, and presented an experimental evaluation of the effectiveness of our non-equivalent generation for a variety of problems with relational constraints. RexGen provides the key step to address the broader problems of synthesis and repair of declarative models in Alloy. The next chapter on AS-

ketch shows how to use the generated expressions to synthesize Alloy models from sketches. We hope our work inspires the development of a broader toolset to support software models and eventually leads to more reliable software systems.

Table 3.4: RexGen performance. Times are in ms. ⊥ indicates a timeout (>1 hour).

Problem		No Pr.		AC Pr.		Static Pr.		Dynamic Pr.		Modulo Pr.	
		#expr	time	#expr	time	#expr	time	#expr	time	#expr	time
addr	4	231	2	199	4	129	25	118	1259	108	279
	5	3984	18	2374	19	1335	56	823	15869	600	1396
	6	7913	27	5563	29	2034	64	1193	19359	900	1879
	7	139971	204	65346	131	24839	189	7116	635296	3546	7902
bempl	4	427	5	377	6	261	29	246	1939	237	343
	5	7027	27	4369	25	2463	64	1708	25098	1424	2999
	6	15396	50	11144	41	4096	80	2588	43840	2198	3814
	7	254843	363	128706	274	47747	296	15363	1555983	10309	29174
btree	4	415	4	355	6	223	29	215	6247	196	484
	5	3264	18	2391	18	1032	51	915	62153	740	1920
	6	17956	42	12919	41	4553	93	3424	999892	2227	6221
	7	139882	204	88578	148	25031	195	⊥	⊥	8505	26140
ctree	4	369	4	327	6	202	27	185	2773	144	754
	5	4625	21	3031	21	1446	59	996	28674	737	5282
	6	14315	37	10707	38	3473	79	2143	192314	1169	9584
	7	168181	221	93805	175	27660	213	⊥	⊥	5530	60968
dijkstra	4	287	2	251	4	140	26	135	2235	133	264
	5	4661	19	2763	20	1397	53	1097	20544	1069	2185
	6	9939	30	7159	39	2175	60	1637	36446	1552	3083
	7	138703	213	65991	139	17976	180	7007	670275	5704	17820
dtree	4	111	1	95	3	40	21	38	680	37	144
	5	581	4	438	6	116	30	105	2809	102	401
	6	2957	15	2130	14	376	39	250	11247	234	841
	7	17109	40	11191	34	1464	61	771	82268	691	1686
farmer	4	1077	8	939	8	654	39	619	28327	454	695
	5	41007	73	24322	53	16969	141	⊥	⊥	5116	8992
	6	96607	140	68468	124	33097	215	⊥	⊥	9247	22555
	7	3666499	7942	1661501	4581	923952	3985	⊥	⊥	80553	2156722
gene	4	641	5	551	6	376	32	348	10853	242	916
	5	12055	31	7653	29	4597	83	3228	632913	1675	8614
	6	42897	76	30703	64	14621	145	⊥	⊥	4324	20490
	7	763031	1998	393015	1150	177920	665	⊥	⊥	26222	326804
grade	4	421	4	373	6	267	30	244	2570	229	447
	5	6533	25	4168	25	2342	65	1496	28995	1105	2450
	6	14930	45	11033	42	4141	89	2321	52542	1740	3446
	7	234482	373	122012	258	45312	311	13166	1858565	7300	19416
hshake	4	471	3	403	5	260	31	244	8543	173	1131
	5	5625	20	3805	22	1936	61	1505	164478	1020	8180
	6	25523	51	18319	46	7640	112	5378	2570827	3031	21775
	7	286661	355	163874	247	58505	318	⊥	⊥	16149	222019
resfm	4	1030	8	940	12	652	38	625	10051	510	767
	5	18692	50	12250	42	7337	99	5705	336197	3626	5634
	6	47128	111	35984	94	13384	146	9406	938031	5026	9076
	7	822434	2107	449935	653	175337	845	⊥	⊥	24997	181425
sll	4	209	2	183	4	104	25	98	1468	98	283
	5	1549	10	1100	13	397	40	331	6868	330	987
	6	6267	25	4694	25	1203	58	808	37988	803	2593
	7	45527	86	28622	64	5463	95	2712	429769	2671	9391

Chapter 4

Solver-based Sketching of Alloy Models[2]

In this chapter, we introduce ASketch – the first solver-based sketching framework for Alloy. We present the technique through an example and show in detail each step of the technique. Experimental results show that ASketch works well for different Alloy models with various number of holes, providing a promising approach to bring the success of traditional program sketching for imperative and functional programs to declarative, relational logic.

The rest of the chapter is organized as follows. Section 4.1 presents an example. Section 4.2 describes the technique in detail. Section 4.3 evaluates ASketch. Section 4.4 summarizes the chapter.

4.1 Example

To illustrate our ASketch approach, consider the following partial Alloy model for an acyclic singly linked list:

```
one sig List { header: lone Node }   sig Node { link: lone Node }
pred Acyclic() { \Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ \E,e\ }
q := {| all|no|some|lone|one |}
```

[2]Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. Solver-based sketching Alloy models using test valuations. In ABZ, 2018.

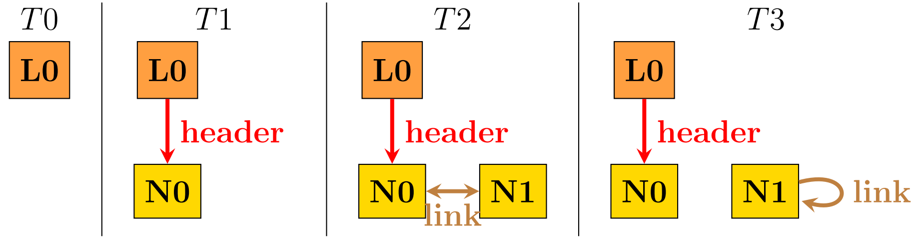

Figure 4.1: Four test valuations shown graphically: $T0$, $T1$, and $T3$ are valid for the expected acyclicity; $T2$ is invalid. $L0$ is the list atom; $N0$ and $N1$ are node atoms.

```
co := {| =|in|!=|!in |}
e := {| (List.header|n).(~?)(*|^)link |}
```

The *signature* (`sig`) declaration introduces a set of atoms and a user-defined type. A signature may declare *fields*, i.e., relations. `List` declares a set of list atoms; `one` makes the set *singleton*, i.e., have exactly 1 atom, which represents the list we are modeling. The field `header` declares a binary relation of type `List` × `Node`; `lone` declares `header` to be a *partial* function, i.e., each `List` atom maps to at most one `Node` atom. `Node` declares a set of nodes and introduces the field `link`, which is a partial function of type `Node` × `Node`. The predicate (`pred`) `Acyclic` introduces a named formula (which may have parameters).

The body of the predicate is a formula *sketch* with three different kinds of holes: `\Q,q\` (quantifier hole), `\CO,co\` (comparison operator hole), and `\E,e\` (expression hole). For the sake of illustrative example, we create several holes of different kinds (potentially more than a user would actually create), and we explicitly list all potential fragments for each hole. Each hole states the

syntactic kind of the hole followed by an identifier, e.g., `E` followed by `e`. Each identifier refers to a regular expression (within `{| ... |}`, following [126]), e.g., `e` refers to `"(List.header|n).(~?)(*|^)link"`, which encodes a set of eight Alloy expressions in this example, including expressions `List.header.*link` and `n.^link`. ASketch extends the Alloy grammar [136] with these holes. The variable `n` is introduced by the quantifier (to be sketched) and is of type `Node`; the operator `=>` denotes logical implication.

The goal is to fill in the holes such that the formula constrains the nodes in the list to form an acyclic structure. Figure 4.1 graphically illustrates four test valuations for the model. Three valuations—$T0$, $T1$, and $T3$—are valid with respect to the expected acyclicity constraint. One valuation, $T2$, is invalid. Note that $T3$ is valid although $N1$ links to itself: $N1$ is not in the list, and the formula we are sketching should constrain only the nodes that are in the list, i.e., reachable from the `header`.

The user can provide the test valuations simply as Alloy predicates. For example, the following represent test valuations $T0$ and $T2$ from Figure 4.1:

```
pred Test0() {
  some L0: List {
    List = L0 and no header and no Node and no link and Acyclic[] }}
pred Test2() {
  some L0: List | some disj N0, N1: Node {
    List = L0 and header = L0->N0
    Node = N0+N1 and link = N0->N1 + N1->N0 and !Acyclic[] }}
```

The predicate `Test0` uses an existentially quantified (`some`) formula to assign a value to the `List` set. Using the Alloy keyword `"no"`, `Test0` declares the other

signatures and relations to be empty. The predicate invocation `Acyclic[]` labels the valuation as *valid* for the expected acyclicity constraint. The predicate `Test2` uses existentially quantified formulas to assign values to the `List` and `Node` sets. The keyword "`disj`" requires the variables in the declaration to represent disjoint sets (i.e., unique nodes), the operator `->` denotes Cartesian product, the operator `+` denotes set union, and the predicate invocation `!Acyclic[]` labels the valuation as *invalid* for the expected acyclicity constraint.

Consider using ASketch to complete all five holes. Two are expression holes \\`E,e`\\ with the same given regular expression assigned for the fragment space, and each expression hole has eight syntactically different expression fragments. Alloy also allows five quantifiers for \\`Q,q`\\ (`all`, `no`, `some`, `lone`, and `one`) and four comparison operators for \\`CO,co`\\ (`=`, `in`, `!=`, and `!in`). In total, there are $5 \times 4 \times 8 \times 4 \times 8 = 5,120$ candidate Alloy models. For our example, we use 8 test valuations to obtain the expected solutions (4 shown in Figure 4.1 plus 4 more). To complete the sketch, ASketch takes less than 1 second when solving the entire Alloy meta-model that encodes all 5,120 models and 8 valuations at once. Here is a solution ASketch finds:

```
all n: Node | n in List.header.*link => n !in n.^link
```

The Alloy keyword "`in`" represents the subset, and `!` denotes logical negation. The operator `*` denotes reflexive transitive closure, and `^` denotes transitive closure. The expression "`List.header.*link`" represents the set of all nodes reachable from the list's header (following *zero* or more traversals

of the field `link`). The expression "`n.~link`" represents the set of all nodes reachable from `n` (following *one* or more traversals of the field `link`). Thus, this universally quantified formula states that for any node that is in the list, the node is not reachable from itself, which correctly characterizes our expected acyclicity constraint.

4.2 ASketch Framework

We next present the ASketch grammar for Alloy models with holes and describe how ASketch determines which fragments complete the sketch to produce an Alloy model that satisfies all the given test valuations.

4.2.1 Input Language

The input to ASketch is an Alloy model with holes. For lack of space, we do not show the full grammar for ASketch's input language, but it effectively extends the Alloy grammar with new syntactic constructs that represent holes. The current Alloy grammar is available at `http://alloy.csail.mit.edu/alloy/documentation/alloy4-grammar.txt`; we follow an older exposition [42] that included the semantics of the kernel Alloy language. Consider this part of the ASketch grammar:

```
quant ::= "all" | "no" | "some" | "lone" | "one" | "\Q," identifier "\"
expr ::= "*"expr | expr "+" expr | ... | "\E," identifier "\"
compareOp ::= "=" | "in" | "!=" | "!in" | "\CO," identifier "\"
formula ::= quant v ":" type "|" formula | ...
regExDecl ::= identifier ":=" "{|" regex "|}"
regex ::= nonSpecial | regex "?" | "(" regex ")" | regex regex | regex "|" regex
```

Table 4.1: Supported fragments for non-recursively defined holes

<table>
<tr><th>Sketch Kind</th><th>Hole</th><th>Candidates</th><th>Sketch Kind</th><th>Hole</th><th>Candidates</th></tr>
<tr><td>Quantifier</td><td>\Q\</td><td>all, no, some, lone, one</td><td>Unary Operator Formula</td><td>\UOF\</td><td>!, ϵ</td></tr>
<tr><td>Logical Operator</td><td>\LO\</td><td>||, &&, <=>, =></td><td>Unary Operator Expression</td><td>\UOE\</td><td>~, *, ^</td></tr>
<tr><td>Compare Operator</td><td>\CO\</td><td>=, in, !=, !in</td><td>Binary Operator</td><td>\BO\</td><td>&, +, -</td></tr>
<tr><td>Unary Operator</td><td>\UO\</td><td>no, some, lone, one</td><td></td><td></td><td></td></tr>
</table>

We extend `quant` so the quantifier can be a hole \`Q,i`\ where `Q` indicates the quantifier hole kind and `i` is an identifier that maps to a regular expression via `regExDecl`. The `expr` options include the expressions from Alloy, formed with unary (e.g., `*`) or binary operators (e.g., `+`), and we add a hole (\`E,i`\) that can replace an entire expression. Comparison operators include all operators from Alloy and also a hole \`CO,i`\. The `formula` options include the Alloy first-order logic formulas. `regExDecl` has the form `i:={|e|}` where `i` is referred from a hole and `e` is a regular expression. We follow the design of popular sketching system [46, 126, 128] that include a few regular expression operators: options (`e?`), concatenations (`e1 e2`), and choices (`e1 | e2`). `nonSpecial` is any character that Alloy supports except for `"?"`, `"("`, `")"`, and `"|"`; to use those, requires escaping them as `"\("`, `"\)"`, and `"\|"`. Finally, ASketch generates all possible fragments that match `e` using a standard backtracking algorithm [65]. ASketch supports all fragments for non-expression holes, as shown in Table 4.1. Our current implementation requires an explicit regular expression for every hole, although a default could be set up such that non-expression holes implicitly get all possible fragments without listing them explicitly.

4.2.2 Solver-based sketching

ASketch reduces the sketching problem to a constraint-solving problem in the Alloy language itself, which is then solved by the Alloy Analyzer. Effectively, ASketch generates one *meta-model* in Alloy that encodes multiple potential solutions (i.e., concrete models) to the sketch. To represent the fragments for each hole, two constructs are added to the meta-model: (1) an Alloy *atom* that names a specific fragment for the hole, and (2) constraints that characterize the semantics of the different fragments for the sketch.

Because ASketch uses the Alloy tool-set itself to encode Alloy expressions and formulas, their semantics need not be explicitly modeled in Alloy; rather, they just need to be stated—indeed, the Alloy tool-set understands the semantics of Alloy. Therefore, we can use a shallow embedding of Alloy fragments in the model. Specifically, to represent the expression fragments, ASketch creates new Alloy *functions*, i.e., parameterized expressions. To represent the operator fragments, ASketch creates new Alloy *predicates*, i.e., parameterized formulas. Moreover, to encode multiple given test valuations in the same meta-model, ASketch *parameterizes* formulas with respect to user-defined relations, which are extracted out of their declaring *signatures* and added as new parameters. Our encoding allows constraining the model with respect to *all* valuation constraints at once—without causing an unnecessary increase in the number of propositional variables in the resulting SAT formula and without requiring higher-order solving [88].

We use the linked-list example from Section 4.1 to describe how AS-

ketch sketches the body of a predicate and completes five holes of three kinds—quantifiers (`\Q,q\`), comparison operators (`\CO,co\`), and expressions (`\E,e\`). ASketch uses the following steps to create an Alloy meta-model whose solutions complete the sketch: (1) parameterize Alloy construct (Section 4.2.2.1); (2) create Alloy meta constructs to encode holes (Section 4.2.2.2); (3) translate test valuations to facts (Section 4.2.2.3); and (4) invoke the Alloy Analyzer to complete the holes (Section 4.2.2.4).

4.2.2.1 Parameterize Alloy constructs

In the first step, ASketch parameterizes all predicates, functions, and facts. To parameterize an Alloy fact, ASketch first converts it to a semantically equivalent predicate. Without loss of generality, we only present how ASketch parameterizes predicates. The goal is to allow *multiple* test valuations to be encoded *in the same meta-model*. ASketch constructs a meta-model which includes (1) all signature declarations from the partial model, but *without* any of the declared relations, and (2) all predicates. Moreover, all predicates in the meta-model get additional parameters: one new parameter per signature and one new parameter per field; parameters that represent signatures have fresh variable names generated, whereas those that represent fields use the same names as in the partial model. In the body of the predicates, any reference to a declared signature is replaced by the corresponding fresh variable name.

For our acyclic linked-list example from Section 4.1, we get:

```
one sig List {}   sig Node {}
pred Acyclic(ls: one List, header: List -> Node,
```

```
  ns: set Node, link: Node -> Node) {
  \Q,q\ n: ns | n \CO,co\ \E,e\ => n \CO,co\ \E,e\ }
```

4.2.2.2 Create Alloy meta constructs to encode holes

ASketch creates Alloy meta constructs that encode concrete values for every hole in Alloy predicates. We present how to encode only quantifier holes, comparison operator holes, and expression holes in Alloy predicates. The algorithm takes as inputs a mapping from expression holes to the corresponding expression fragments and a mapping from holes to *all Alloy variables* (sigs, fields, predicate parameters, `let`-bound variables, and quantified variables) in scope of the holes. The algorithm iterates over each Alloy predicate in the meta-model and updates the predicate body by recursively replacing ASketch holes with predicate/function calls, and creating and adding the predicate/function declarations to the meta-model. Note that any reference to a declared signature in the generated predicate/function is replaced by the corresponding fresh variable name as described in Section 4.2.2.1, e.g., `List` with `ls`.

After this step, ASketch constructs the following meta-model (note that the two comparison operator holes share the same operator fragments, and the two expression holes share the same expression fragments):

```
pred Acyclic(ls: one List, header: List -> Node,
  ns: set Node, link: Node -> Node) {
  q1[RQ1, ls, header, ns, link] }
abstract sig Q {}   one sig RQ1 in Q {}
one sig Q_All, Q_No, Q_Some, Q_Lone, Q_One extends Q {}
pred q1(h: Q, ls: one List, header: List -> Node,
  ns: set Node, link: Node -> Node) {
  h = Q_All => all n: ns | co2[RCO2, n, expr3[RE3, ls, header, ns, link, n]]
```

```
    => co2[RCO4, n, expr3[RE5, ls, header, ns, link, n]]
  h = Q_No => no n: ns | co2[RCO2, n, expr3[RE3, ls, header, ns, link, n]]
    => co2[RCO4, n, expr3[RE5, ls, header, ns, link, n]]
  ... }
abstract sig CO {}   one sig RCO2 in CO {}   one sig RCO4 in CO {}
one sig CO_Eq, CO_In, CO_NEq, CO_NIn extends CO {}
pred co2(h: CO, e1, e2: set univ) {
  h = CO_Eq => e1 = e2
  h = CO_In => e1 in e2
  ... }
abstract sig E3 {}   one sig RE3 in E3 {}   one sig RE5 in E3 {}
one sig E3_1, E3_2, E3_3, E3_4, E3_5, E3_6, E3_7, E3_8 extends E3 {}
fun expr3(h: E3, ls: one List, header: List -> Node,
          ns: set Node, link: Node -> Node, n: one Node): univ {
  (h = E3_1 => ls.header.*link else
  (h = E3_2 => n.^link else
  ... else none)) }
```

For quantifier holes, ASketch creates a unique *abstract* sig `Q` and declares 5 disjoint singleton sigs that represent all possible values for the hole (`all`, `no`, `some`, `lone`, and `one`). For each quantifier hole, ASketch translates the quantified formula to a predicate call. The predicate has the following parameters: (1) one parameter of the new abstract sig type that allows evaluating the predicate to one of the 5 quantifiers; and (2) one parameter for each variable in scope: signatures and fields from the original model, and optionally, predicate parameters, `let`-bound variables, and/or quantified variables in case of nested quantified formulas. The corresponding predicate declaration, `q1` in our example, is added to the meta-model. The predicate body is a conjunction of implications that model different quantified formulas corresponding to the hole. ASketch also introduces a result sig, `RQ1` in our example, that will obtain one of the 5 values (`Q_All`, `Q_No`, `Q_Some`, `Q_Lone` and `Q_One`) to represent the quantifier to fill in the hole.

For comparison operator holes, ASketch creates a unique *abstract* sig `CO` and declares 4 disjoint singleton sigs that represent all possible values for the hole (`=`, `in`, `!=`, and `!in`). Unlike for quantifier holes where each hole requires a new predicate, all comparison operator holes (of the same arity) can be encoded using a single predicate if they share the same set of fragments. ASketch creates a predicate, `co2` in our example, which encodes a formula that contains a comparison operator. The predicate contains 3 parameters: (1) one parameter of the new abstract sig type that allows evaluating the predicate to one of the 4 comparison operators (`CO_Eq`, `CO_In`, `CO_NEq`, and `CO_NIn`); (2) left operand; and (3) right operand. For each comparison operator hole, ASketch introduces a result sig, `RCO2` and `RCO4` in our example, similar as for quantifier holes. (ASketch treats the other non-expression holes similar to comparison operator holes.)

To model values of expression holes, ASketch creates one new *abstract* sig, `E3` in our example, for all holes that share the same set of expression fragments and declares k singleton sigs that partition the new sig, where k is the number of expression fragments for the corresponding expression hole, 8 in our example. ASketch also introduces result sigs, `RE3` and `RE5` in our example, that will obtain one of the k values to represent which fragment fills the hole. Next, ASketch creates an Alloy function that can select from these choices. The function has these parameters: (1) one parameter of the new abstract sig type that allows evaluating the function to one of the expression fragments based on the invocation context; and (2) one parameter for each Alloy variable

in scope. The function body is a nested if-then-else expression where exactly one choice is true for any invocation, and the function evaluates to the value of the expression fragment corresponding to that choice.

4.2.2.3 Express test valuations as facts

To complete the sketch with respect to the given test valuations (labeled as valid or invalid), ASketch automatically translates the test valuations (expressed as predicates in Section 4.1) to *facts*, which forces any solution that is created (in the final meta-model) to conform to all given valuations. Because valuations from different tests may contradict one another, ASketch uses Alloy's `let` construct to introduce the necessary names for sets and relations that are assigned values. Then, ASketch passes these sets and relations to the parameterized predicates (described in Section 4.2.2.1) so that the final sketched model satisfies all the tests at once. For example, `Test0` from Section 4.1 becomes the following fact:

```
fact Test0 {
  some L0: List {
    let ls = L0 | let header = none->none |
    let ns = none | let links = none->none |
      Acyclic[ls, header, ns, links] }}
```

4.2.2.4 Invoke Alloy Analyzer to complete holes

The final meta-model consists of all pieces generated in sections 4.2.2.1, 4.2.2.2, and 4.2.2.3. ASketch invokes the Alloy Analyzer to execute an empty `run` command (`run {}`) on the final meta-model. The analyzer searches for

possible valuations of the result `R` sigs so that they conform to all tests. In our example, `RQ1` evaluates to `Q_All`, `RCO2` to `CO_In`, `RE3` to `E3_1`, `RCO4` to `CO_NIn`, and `RE5` to `E3_2`. Finally, ASketch maps result values to the corresponding Alloy fragments and reports concrete values of all holes to the user, e.g., ⟨`all, in, List.header.*link, !in, n.^link`⟩ in our example. The completed, sketched model becomes this:

```
one sig List { header: lone Node }   sig Node { link: lone Node }
pred Acyclic() { all n: Node | n in List.header.*link => n !in n.^link }
```

Our example used only 8 expressions, but realistic ASketch models may have hundreds of expressions, which results in much larger meta-models. Our experiments show that the above encoding technique still works relatively well even for a large number of expressions. It also works much better than all other meta-model encoding techniques we tried.

4.3 Evaluation

We next present our experimental evaluation of ASketch. We use five small but intricate Alloy problems to derive 24 sketching models for evaluation (Section 4.3.1). We evaluate how much time ASketch takes to find complete Alloy models that satisfy all test valuations (Section 4.3.2).

4.3.1 Sketching problems

We use 24 sketches derived from five core Alloy models: **sll** from Section 4.1, **btree** models the acyclicity constraint of a binary tree, **contains**

checks whether a list contains an element, **remove** models removing an element from a list, and **dijkstra** models Dijkstra's mutual exclusion algorithm.

For each core model, we picked one predicate to create several sketches by increasing the total number of holes in the body of the predicate, from left to right. This process enables us to systematically create model variants to explore how the number of holes affects our techniques. For example, for **sll**, we identified 3 non-expression holes and 2 expression holes in the `Acyclic` predicate and produced these 5 variants:

```
\Q,q\ n: Node | n in List.header.*link => n !in n.^link       // LinkedList 1H
\Q,q\ n: Node | n \CO,co\ List.header.*link => n !in n.^link  // LinkedList 2H
\Q,q\ n: Node | n \CO,co\ \E,e\ => n !in n.^link              // LinkedList 3H
\Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ n.^link          // LinkedList 4H
\Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ \E,e\            // LinkedList 5H
```

Our experiments are performed on a MacBook Pro running OS X El Capitan with 2.5 GHz Intel Core i7-4870HQ and 16GB of RAM.

4.3.2 ASketch results

Table 4.2 shows the results of ASketch for various sketching problems. The column **Model** shows the model variants for each core model; columns #**N** and #**E** show the number of non-expression holes and expression holes, respectively; the column **Search Space** shows the number of fragments combinations for all holes; and the columns #**PrimVar**, #**Clauses**, and **Solving Time** show the number of primary variables, clauses, and solving time in seconds for the meta model, respectively. The **Search Space** is computed as

the product of the number of fragments for each hole in the model. For example, if the **sll** model with 5 holes has 1 quantifier hole with 5 fragments, 2 comparison operator holes with 4 fragments each, and 2 expression holes with 400 fragments each, then the sketching problem has a search space of $5 \times 4^2 \times 400^2 = 12,800,000 \cong 1.3e7$.

The columns 50, 100, 200, 300, and 400 show the number of expression fragments in the experiment, e.g., 50 means that we use 50 syntactically different expressions for each expression hole in the model variant. We generate regular expressions for expression holes using RexGen such that two properties hold. First, the set of expressions contains the expected solutions. Second, the larger set of expressions contains all expressions from the smaller set, e.g., the set of 100 expressions includes the set of 50 expressions and adds 50 more. We ensure the first property as follows. Suppose we have H expression holes and E expected expressions to fill the holes. We run RexGen to get X expressions and exclude E expected expressions from X expressions. Next, we run ASketch to find all solutions w.r.t. the test valuations and exclude any expression in the solutions that is non-equivalent to any of the E expected expressions. The idea is to remove all expressions that could lead to a solution that passes all tests but is incorrect. Then, to form a set of expressions with size Y (where Y is 50, 100, 200, 300, or 400), we sample the remaining expressions to obtain $Y - E$ expressions, and add the E expected expressions back .

dijkstra has two expression holes with different variables in scope, so each expression hole uses a different set of expression fragments (but with

the same number of expressions). Expression holes for each of **sll**, **btree**, **contains**, and **remove** share the same set of expression fragments. In the experiments, we use 16 test valuations for each core model, and all model variants of the same core model share the same test suite. All experiment settings, with various fragments and test valuations, yield solutions that are semantically equivalent to the correct solutions.

If a sketch has no expression hole, then increasing the number of the expression fragments does not increase the search space, primary variables, or clauses in the generated meta-model. For example, **btree** model with 1 hole has only a comparison operator hole, and the search space (4), the number of primary variables (170), and clauses (7,957) remain unchanged as the number of expression fragments increases. If the sketch has expression holes, then the search space, primary variables, and clauses increase when we use more expression fragments. In our experiment, the search space goes up to 4.1e9 (**btree**), the number of primary variables goes up to 1420 (**remove**), and the number of clauses goes up to 2.3e6 (**dijkstra**). Overall these numbers show that the sketching problems are non-trivial.

The solving time depends on various factors, including the number of primary variables and clauses, the size of each clause, the complexity of the expression fragments, the search strategy of the SAT solver, etc. In general, the solving time increases with the size of the search space and the number of holes. However, there are exceptions. For example, in **sll** with 4 holes, the solving time decreases as the size of expression fragments grows from 300 to 400. The

Table 4.2: ASketch results for finding a solution. Times are in seconds.

Model		#N	#E	Search Space					#PrimVar					#Clauses					Solving Time				
				50	**100**	**200**	**300**	**400**	**50**	**100**	**200**	**300**	**400**	**50**	**100**	**200**	**300**	**400**	**50**	**100**	**200**	**300**	**400**
sll	1H	1	0	5	5	5	5	5	138	138	138	138	138	6170	6170	6170	6170	6170	0.1	0.2	0.1	0.1	0.1
	2H	2	0	20	20	20	20	20	142	142	142	142	142	7397	7397	7397	7397	7397	0.2	0.2	0.2	0.2	0.2
	3H	2	1	1e3	2e3	4e3	6e3	8e3	192	242	342	442	542	6.3e4	1.1e5	2.3e5	3.6e5	5.1e5	2.8	4.5	88.6	267.9	141.0
	4H	3	1	4e3	8e3	1.6e4	2.4e4	3.2e4	196	246	346	446	546	6.5e4	1.2e5	2.4e5	3.8e5	5.3e5	3.2	5.2	91.1	286.6	150.1
	5H	3	2	2e5	8e5	3.2e6	7.2e6	1.3e7	246	346	546	746	946	1.1e5	2.1e5	4.4e5	7e5	1e6	6.7	32.5	252.9	574.6	759.1
btree	1H	1	0	4	4	4	4	4	170	170	170	170	170	8e3	8e3	8e3	8e3	8e3	0.1	0.1	0.1	0.1	0.1
	2H	1	1	2e2	4e2	8e2	1200	1600	220	270	370	470	570	6e4	1.1e5	2.2e5	3.5e5	4.8e5	0.6	1.1	3.1	5.7	5.6
	3H	2	1	8e2	1600	3200	4800	6400	224	274	374	474	574	6.1e4	1.1e5	2.2e5	3.5e5	4.9e5	0.5	1.4	2.9	3.6	12.3
	4H	2	2	4e4	1.6e5	6.4e5	1.4e6	2.6e6	274	374	574	774	974	8.6e4	1.7e5	3.6e5	5.8e5	8.4e5	1.4	5.6	27.6	36.9	265.7
	5H	3	2	1.6e5	6.4e5	2.6e6	5.8e6	1e7	278	378	578	778	978	8.6e4	1.7e5	3.6e5	5.8e5	8.4e5	1.8	7.0	30.3	47.5	127.4
	6H	3	3	8e6	6.4e7	5.1e8	1.7e9	4.1e9	328	478	778	1078	1378	1.1e5	2.3e5	4.9e5	8.1e5	1.2e6	1.4	37.7	138.7	515.1	709.3
contains	1H	0	1	50	1e2	2e2	3e2	4e2	312	362	462	562	662	3.3e4	6.1e4	1.3e5	2.2e5	3.2e5	0.4	0.7	3.2	7.4	7.6
	2H	1	1	2e2	4e2	8e2	1200	1600	316	366	466	566	666	5.4e4	1e5	2.1e5	3.2e5	4.6e5	0.7	1.7	9.4	25.1	24.9
	3H	1	2	1e4	4e4	1.6e5	3.6e5	6.4e5	366	466	666	866	1066	7.7e4	1.5e5	3.3e5	5.4e5	7.9e5	2.0	15.6	103.3	397.0	1925.2
remove	1H	0	1	50	1e2	2e2	3e2	4e2	267	317	417	517	617	4e4	7.3e4	1.5e5	2.5e5	3.6e5	0.2	0.5	1.1	2.2	4.4
	2H	1	1	150	3e2	6e2	9e2	1200	270	320	420	520	620	4.1e4	7.4e4	1.5e5	2.5e5	3.6e5	0.3	0.7	1.3	2.8	5.3
	3H	1	2	7500	3e4	1.2e5	2.7e5	4.8e5	320	420	620	820	1020	7.2e4	1.4e5	3e5	5e5	7.3e5	0.7	2.0	3.6	16.7	75.9
	4H	1	3	3.8e5	3e6	2.4e7	8.1e7	1.9e8	370	520	820	1120	1420	1e5	2.1e5	4.5e5	7.5e5	1.1e6	7.0	62.2	61.6	6903.8	19579.8
dijkstra	1H	1	0	4	4	4	4	4	370	370	370	370	370	9348	9348	9348	9348	9348	0.1	0.1	0.0	0.0	0.0
	2H	1	1	2e2	4e2	8e2	1200	1600	420	470	570	670	770	9.2e4	1.4e5	2.6e5	3.9e5	5.4e5	1.1	2.1	4.8	24.2	21.9
	3H	2	1	1e3	2e3	4e3	6e3	8e3	404	454	554	654	754	9.2e4	1.4e5	2.6e5	3.9e5	5.4e5	1.1	2.2	4.9	21.6	22.1
	4H	3	1	5e3	1e4	2e4	3e4	4e4	409	459	559	659	759	9.4e4	1.5e5	2.6e5	4e5	5.4e5	1.9	4.2	6.0	40.9	41.3
	5H	4	1	2e4	4e4	8e4	1.2e5	1.6e5	413	463	563	663	763	9.8e4	1.5e5	2.7e5	4e5	5.5e5	1.7	2.9	14.3	26.9	46.9
	6H	4	2	1e6	4e6	1.6e7	3.6e7	6.4e7	463	563	763	963	1163	2.7e5	5.4e5	1.1e6	1.7e6	2.3e6	26.8	80.6	1053.1	4542.8	6535.0

reason is that multiple expression fragments are correct and equivalent. We cannot control how the Alloy Analyzer generates CNF clauses from the meta-model, so some solutions are found sooner than the others even if we increase the search space. Another exception is when **btree** goes from 4 holes to 5 holes using 400 expression fragments. Again, the solving time decreases as the number of holes increases. The reasons are that (1) adding an operator hole does not increase the number of primary variables or clauses by much; (2) it can make the sketching problem easier to solve as more equivalent correct solutions can be found; and (3) the Alloy Analyzer encodes the problem such that the solver is able to find the solution fast. Overall, ASketch's encoding is relatively efficient and works well for large search spaces.

4.4 Summary

We introduced ASketch, the first solver-based approach for sketching Alloy models. Given a model with holes and some (valid and invalid) valuations for the desired model, ASketch completes the given model with respect to the valuations. ASketch performs two key steps: it generates a pool of fragments (e.g., expressions) for each hole from user-provided regular expressions, and it creates a meta-model to explore the resulting space of candidate (completed) models to find a model that conforms to the valuations. An experimental evaluation using a suite of sketches shows that ASketch introduces a promising approach for sketching Alloy models. ASketch brings the spirit of traditional program sketching [7, 38, 46, 60, 123, 125–129]—often regarded as

the breakthrough approach in program synthesis for imperative and functional programs during the last decade—to a declarative, relational logic. We hope ASketch serves as a sound basis for a highly effective methodology for synthesizing Alloy models, which ultimately increases the use of analyzable models and leads to better software.

Chapter 5

Fault Localization for Alloy

In this chapter, we introduce AlloyFL, which includes a set of fault localization techniques at the AST node granularity for Alloy. These techniques include the spectrum-based (AlloyFL$_{co}$), mutation-based (AlloyFL$_{mu}$ and AlloyFL$_{hy}$) and sat-based (AlloyFL$_{un}$ and AlloyFL$_{su}$) fault localization techniques. We also introduce new distance metrics that measure the cost for human to inspect the list of suspicious nodes returned by AlloyFL, following the spirit of the nearest neighbor distance metric based on program dependence graphs [111]. The experimental results show that mutation-based fault localization techniques outperform other techniques.

The rest of the chapter is organized as follows. Section 5.1 presents an example. Section 5.2 describes in detail the techniques to locate faults. Section 5.3 describes the metrics we introduce to measure AlloyFL. Section 5.4 evaluates AlloyFL. Section 5.5 summarizes the chapter.

5.1 Example

This section presents a real-world faulty Alloy model to introduce key concepts for AlloyFL. We briefly describe the basics of Alloy and AUnit as

```
open util/ordering[State] as ord
abstract sig Object { eats: set Object }
one sig Farmer, Fox, Chicken, Grain
  extends Object {}
fact eating {
  eats = Fox->Chicken + Chicken->Grain  }
sig State { near,far: set Object }
fact initialState {
  let s0 = ord/first |
    s0.near = Object && no s0.far  }
pred crossRiver[from,from',to,to': set Object] {
  (from' = from - Farmer &&
    to' = to - to.eats + Farmer )
  || (some item: from - Farmer {
  from' = from - Farmer - item &&
    to' = to - to.eats + Farmer + item  }) }
fact stateTransition {
  all s: State, s': ord/next[s] {
    Farmer in s.near =>
      crossRiver[s.near,s'.near,s.far,s'.far]
    else
      crossRiver[s.far,s'.far,s.near,s'.near]}
}
pred solvePuzzle { ord/last.far = Object }
```

(a) Faulty Farmer Model

```
pred test1 {
  some disj F0: Farmer |
  some disj X0: Fox |
  some disj C0: Chicken |
  some disj G0: Grain |
  some disj F0, X0, C0, G0: Object |
  some disj S0, S1, S2, S3: State {
    Farmer = F0
    Fox = X0
    Chicken = C0
    Grain = G0
    Object = F0 + X0 + C0 + G0
    eats = X0->C0 + C0->G0
    State = S0 + S1 + S2 + S3
    near = S0->F0 + S0->X0 + S0->C0 + S0->G0 +
      S1->X0 + S2->F0 + S2->X0 + S3->X0
    far = S1->F0 + S1->G0 + S2->G0 +
      S3->F0 + S3->G0
    ord/first = S0
    ord/next = S0->S1 + S1->S2 + S2->S3
    crossRiver[F0+X0+C0+G0,C0,none,F0+X0]
  } }
run test1 for 4 expect 1
// More tests ...
```

(b) MuAlloy Generated Tests

Figure 5.1: Faulty Farmer Example and MuAlloy Generated Tests.

needed.

Figure 5.1a shows a faulty Alloy model of the well-known "farmer river-crossing" puzzle where the goal is to allow a farmer to transport a chicken, fox, and grain from one river bank to the other on a boat. However, the farmer can only carry one belonging on the boat at a time, and if left unattended, the fox will eat the chicken and the chicken will eat the grain. The model

contains a modeling error which prevents the "eating" from happening while the farmer is away, and instead requires the farmer to be back. Figure 5.1b shows an AUnit test that fails.

The *signature* (`sig`) declaration, "`sig Object`", introduces a set of object atoms; `abstract` means the `Object` signature cannot have atoms of its own type, but its subsignatures can have atoms. `eats` is a *relation* that maps an `Object` atom to a set of `Object` atoms. `Farmer`, `Fox`, `Chicken` and `Grain` are declared as singleton subsignatures of `Object`. The *fact* `eating` states that the fox eats the chicken and the chicken eats the grain. Note that any *fact* in Alloy is enforced to be true. Signature `State` models the objects in both the near and far banks after every farmer's cross-river movement. The `open` declaration linearly orders the `State` atoms. The fact `initialState` constrains that initially everything is on the near bank and nothing is on the far bank. The *predicate* `crossRiver` defines the river crossing action. It takes four parameters (2 pairs of pre and post states): the set of objects on the bank where the farmer starts at (pre-state:`from` and post-state:`from'`) and crosses to (pre-state:`to` and post-state:`to'`) before and after the cross-river movement. The predicate states that either the farmer takes nothing or the farmer takes one item not including himself to the other side of the river. For the case when the farmer takes nothing the model uses a conjunction formula: "`from' = from - Farmer && to' = to - to.eats + Farmer`" which means that after the farmer crosses the river from bank, say A to bank, say B, the farmer is removed from the set of objects on bank A and added to the set of objects on bank B. When

moving, the objects on bank B could change because an object may eat another object before the farmer's arrival. For the case when the farmer takes an item, the model uses an existentially quantified formula: "`some item: from - Farmer | ...`". The fact `stateTransition` states that for every two consecutive states, if the farmer is on the near bank in the pre-state, then he would cross the river to the far bank. Otherwise, he would cross the river from the far bank to the near bank. The predicate `solvePuzzle` restricts that in the last state, everything should be on the far bank.

The faults are in the predicate `crossRiver` and are colored in orange. The predicate considers eating to happen in "`to`" instead of "`from`", which stops the farmer from leaving and letting the fox eat the chicken without the farmer coming back. The correct formula should be: "`from' = from - Farmer - from'.eats && to' = to + Farmer`" and "`from' = from - Farmer - item - from'.eats && to' = to + Farmer + item`". This modeling error is introduced in Alloy release 4.1 and fixed in release 4.2. An automatically generated AUnit test that reveals the fault is shown in Figure 5.1b. Predicate `test1` encodes the valuation of each signature type in the farmer model in Figure 5.1a. Each relation is assigned some atoms, e.g. `Farmer` contains a single atom `F0`. The invocation of `crossRiver` predicate states that all objects are on the near bank in the pre-state and nothing (`none`) is on the far bank. In the post-state (after the farmer crosses the river with the fox), only the chicken is left on the near bank (because the chicken is supposed to eat the grain) and both the farmer and the fox are on the far bank. The command "`run test for 4 expect 1`"

runs the test with a scope of at most 4 atoms for each signature type and expects the existence of a solution. However, the faulty farmer model does not have any solution with respect to this test, which contradicts the expectation and causes a test failure.

We use a test suite that contains some failing tests (e.g. `test1`) to locate the fault using AlloyFL. AlloyFL$_{co}$ assigns all Alloy paragraphs equal suspiciousness scores (except the fact `solvePuzzle` as it is never covered by any failing tests) because all tests implicitly invoke facts and the `stateTransition` fact invokes the `crossRiver` predicate. The most suspicious AST nodes are highlighted in red (including yellow and green) in Figure 5.1a. Both AlloyFL$_{un}$ and AlloyFL$_{su}$ report the entire body of the `crossRiver` predicate as the most suspicious AST node which is highlighted in yellow (including green) . AlloyFL$_{mu}$ and AlloyFL$_{hy}$ report the node "`from' = from - Farmer - item && to' = to - to.eats + Farmer + item`" as the most suspicious node because mutating the root node `&&` makes the most failing tests pass compared to mutating other AST nodes. Thus, the most suspicious node `&&` and its descendants are highlighted in green. We can see that the most suspicious node returned by AlloyFL$_{mu}$ and AlloyFL$_{hy}$ is closest to the faulty node. Thus, the MBFL techniques are most accurate among all techniques in this example.

5.2 Technique

In this section, we describe the formulas to compute suspiciousness scores (Section 5.2.1) and all techniques in AlloyFL (Section 5.2.2).

<table>
<tr><th>Name</th><th>Formula</th></tr>
<tr><td>Tarantula [52]</td><td>$\frac{\frac{failed(e)}{totalfailed}}{\frac{failed(e)}{totalfailed}+\frac{passed(e)}{totalpassed}}$</td></tr>
<tr><td>Ochiai [1]</td><td>$\frac{failed(e)}{\sqrt{totalfailed \times (failed(e)+passed(e))}}$</td></tr>
<tr><td>Op2 [95]</td><td>$failed(e) - \frac{passed(e)}{totalpassed+1}$</td></tr>
<tr><td>Barinel [3]</td><td>$1 - \frac{passed(e)}{passed(e)+failed(e)}$</td></tr>
<tr><td>DStar [151]</td><td>$\frac{failed(e)^*}{passed(e)+(totalfailed-failed(e))}$</td></tr>
<tr><td colspan="2">totalfailed: the total number of test cases that failed.
totalpassed: the total number of test cases that pass.
failed(e): the number of failed test cases that cover or kill e.
passed(e): the number of passed test cases that cover or kill e.</td></tr>
</table>

Figure 5.2: Suspiciousness Formulas in AlloyFL.

5.2.1 Suspiciousness Formulas

Figure 5.2 shows the formulas that AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$ support to compute the suspiciousness score. For AlloyFL$_{co}$, the code elements (e) are AST nodes. For AlloyFL$_{mu}$ and AlloyFL$_{hy}$, killed mutants are treated as covered code elements while live mutants are treated as uncovered code elements. *totalfailed* and *totalpassed* are the number of test cases which failed and passed w.r.t. the original model. *failed(e)* and *passed(e)* are the number of test cases which failed and passed that cover the AST node or kill the mutant e.

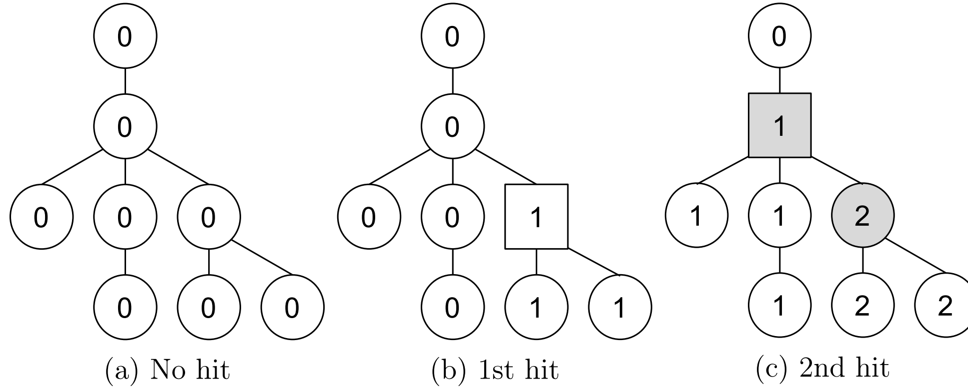

Figure 5.3: Illustration of AlloyFL$_{un}$ and AlloyFL$_{su}$

5.2.2 AlloyFL

AlloyFL locates faults at the AST node granularity, which allows it to locate faulty expressions or formulas that are hierarchical. We present AlloyFL$_{co}$ as the first baseline technique and expect it to be inaccurate because of the non-existence of control flow in the Alloy language. We next present two baseline techniques, i.e. AlloyFL$_{un}$ and AlloyFL$_{su}$ which simulate what Alloy users can achieve by using the unsat core. Finally, we present two more advanced techniques AlloyFL$_{mu}$ and AlloyFL$_{hy}$ which define a diverse set of mutation operators and are shown to be more accurate.

AlloyFL$_{co}$. Since Alloy does not have control-flow and execution traces, every code element in the same paragraph will be either executed together or not executed at all by a given test. This means nodes declared in the same paragraph would share the same suspiciousness score. To implement AlloyFL$_{co}$, we build a static analyzer which analyzes the entire AST and binds a variable

usage or a predicate/function call to its signature or predicate/function declaration. The static analyzer is used to find all Alloy paragraphs transitively used by a test. However, the analyzer ignores dependencies that are never used. For example, if a test uses a formula "`all s: S, t: T | some s && p[s]`" where variable "`t`" is not used and "`p[s]`" is a predicate invocation, then the test only depends on signature "`S`" and predicate "`p`". By default, all facts are implicitly used, and all paragraphs transitively invoked in the facts and the test predicate are covered by the test. AlloyFL$_{co}$ computes a suspiciousness score for each Alloy paragraph based on the number of passing/failing tests that cover it and a formula shown in Figure 5.2. Finally, all paragraphs are ranked in descending order of suspiciousness score. In case of a tie, the paragraph which has a smaller number of descendants is prioritized.

AlloyFL$_{un}$. To implement AlloyFL$_{un}$, we modify the standard Alloy toolset to return AST nodes when the MiniSat solver with unsat core is used [137, 138]. We configure the solver such that it is guaranteed to return a local minimum core and all formulas are fully expanded (pushing negations in as much as possible, removing existential quantifiers using skolemization and expanding universal quantifiers given the bounds on the signatures) to make the returned core as fine-grained as possible. AlloyFL$_{un}$ constructs a hit-map for the entire AST and every node in the AST has a count initially set to 0. If a node is returned by the unsat core, then the count for the node itself *and* each of its descendant increases by 1. To illustrate, Figure 5.3 shows how the hit-map is built. Initially, each node has a count of 0 (Figure 5.3(a)). In Figure 5.3(b),

a node denoted by the square is returned by the unsat core and AlloyFL$_{un}$ increases the counts of all the affected descendants. This process applies for all the subsequently returned nodes. For example, suppose the square node in Figure 5.3(c) is returned next, the count of each descendant is increased to 1 and the count of each previously hit node is increased to 2. Note that a child node always has a count greater than or equal to its parent's count. AlloyFL$_{un}$ collects every node whose count is greater than its parent's count, e.g. the gray nodes in Figure 5.3(c). AlloyFL$_{un}$ does not collect the root node as we set the root's parent to null. The collected nodes are ranked in descending order of the corresponding count. In case of a tie, nodes with a smaller number of descendants are prioritized. Note that AlloyFL$_{un}$ only works for unsatisfiable tests and cannot be used if the model is strictly underconstrained, in which case no unsatisfiable failing test exists.

AlloyFL$_{su}$. Similar to AlloyFL$_{un}$, AlloyFL$_{su}$ also constructs a hit-map for the entire AST. The difference is that AlloyFL$_{su}$ uses nodes reported from both unsatisfible and satisfiable failing tests. The nodes reported from the unsatisfible failing tests are the same as for AlloyFL$_{un}$, and the nodes reported from the satisfiable failing tests are from the static analyzer described for AlloyFL$_{co}$. We give the official algorithm for AlloyFL$_{su}$ in Algorithm 13. The algorithm takes as input a faulty model M and a test suite T, and returns the ranked list of the suspicious AST nodes L. L' keeps the nodes returned by the static analyzer and the unsat core. Both L and L' are initialized to empty lists. The algorithm collects the test results R by invoking T over M. For each

individual test result r, we skip if r is passed. If r fails and is satisfiable, then we collect all transitively used nodes of the corresponding test by invoking the static analyzer and add those nodes to L'. If r fails and is unsatisfiable, we collect all nodes returned by the unsat core and add them to L'. Note that L' is a list of sets of nodes. To sort the nodes, we first initialize a *hitmap* as an empty map with a default value of 0. For every set of *nodes* in L', we increase the counts of each individual node n in *nodes* and n's descendants in the *hitmap*. Then, for each node n whose count is bigger than its parent's count, we add it to L. Finally, we sort L in descending order of the number of times a node is hit and prioritize nodes with a smaller number of descendants in case of a tie. Algorithm 13 boils down to AlloyFL$_{un}$ if we do not collect nodes when the test is satisfiable. The intuition of the algorithm is that nodes covered by more failing tests are more likely to be faulty, and we use nodes returned by the unsat core if possible because the core typically gives finer grained nodes compared to the static analyzer.

***AlloyFL$_{mu}$*.** AlloyFL$_{mu}$ implements a wide variety of mutation operators as shown in Figure 2.2. *MOR* mutates signature multiplicity, e.g. `"one sig"` to `"lone sig"`. *QOR* mutates quantifiers, e.g. `some` to `all`. *UOR*, *BOR* and *LOR* define operator replacement for unary, binary and formula list operators, respectively. For example, *UOR* mutates `a.^b` to `a.*b`; *BOR* mutates `a<=>b` to `a=>b`; and *LOR* mutates `a||b` to `a&&b`. *UOI* inserts an unary operator before expressions, e.g. `a.b` to `a.~b`. *UOD* deletes an unary operator, e.g. `a.^~b` to `a.^b`. *LOD* deletes an operand of a logical operator, e.g. `a&&b` to `b`.

Algorithm 1: Sat-Unsat Based Fault Localization

Input: Faulty Alloy model M, test suite T.
Output: Ranked list of suspicious AST nodes L.

1 $L \leftarrow []$, $L' \leftarrow []$, R = runTests(M, T)
2 **foreach** $r \in R$ **do**
3   **if** *r.isPassed()* **then continue**
4   **if** *r.isSatisfiable()* **then** L'.add(staticAnalyze(r))
5   **else** L'.add(unsatCore(r))
6 *hitmap* $\leftarrow$ <Node,Int>{} // Default value is 0
7 **foreach** *nodes* $\in L'$ **do**
8   **foreach** $n \in$ *nodes* **do**
9   **foreach** $d \in$ *n.getDesc()* **do** *hitmap*[d] += 1
10 **foreach** $n \in$ *hitmap* **do**
11   **if** *hitmap[n.getParent()]* < *hitmap[n]* **then** L.add(n)
12 L.sortByHitAndSize(*hitmap*, *reverse*=**True**)
13 **return** L

PBD deletes the body of an Alloy paragraph. *BOE* exchanges operands for a binary operator, e.g. `a-b` to `b-a`. *IEOE* exchanges the operands of `imply-else` operation, e.g. "`a => b else c`" to "`a => c else b`".

Algorithm 16 shows the details of AlloyFL$_{mu}$. The algorithm takes as input a faulty Alloy model M, a test suite T, a set of mutation operators *Ops* and a suspiciousness formula F. The output of the algorithm is a ranked list of suspicious AST nodes (L) sorted in the descending order of suspiciousness. Initially, L is set to an empty list. S keeps the set of nodes covered by failing tests and is initialized as an empty set. AlloyFL$_{mu}$ runs T against M and the results are stored in R. *n2s* keeps the mapping from a node to its suspiciousness score and it is initialized to an empty map with a default value of 0. For each

Algorithm 2: Mutation-Based Fault Localization

```
Input: Faulty Alloy model M, test suite T, mutation operators
       Ops, suspiciousness formula F.
Output: Ranked list of suspicious AST nodes L.
1  L ← [], S ← ∅, R = runTests(M, T)
2  n2s ← <Node, Double>{} // Default value is 0.0
3  foreach r ∈ R do
4  |   if r.isPassed() then continue
5  |_  foreach n ∈ staticAnalyze(r) do  S.addAll(n.getDesc())
6  foreach n ∈ M.getNodes() do
7  |   if n ∉ S then continue
8  |   foreach op ∈ Ops do
9  |   |   if !isApplicable(op, n) then continue
10 |   |   M' = applyOp(op, n, M)
11 |   |   if isValid(M') && !isEquivalent(M, M') then
12 |   |   |   R' = runTests(M', T)
13 |   |   |_  n2s[n] = max(n2s[n], computeSusp(F, R, R'))
   |   |_
14 |_  if n2s[n] > 0 then L.add(n)
15 L.sortByScore(n2s, reverse=True)
16 return L
```

test result r in R, AlloyFL$_{mu}$ collects nodes and their descendants covered by all failing tests. Then, AlloyFL$_{mu}$ iterates over each node n in M. If n is not covered by any failing test, i.e. $n \notin S$, then AlloyFL$_{mu}$ skips it. For each n covered by the failing tests, AlloyFL$_{mu}$ tries to apply every mutation operator in Ops to the node, one at a time. If the mutation operator is not applicable, it is skipped. Otherwise, AlloyFL$_{mu}$ mutates M to M'. If M' leads to a compilation error or is equivalent to M, then AlloyFL$_{mu}$ skips M'. Otherwise, AlloyFL$_{mu}$ runs T against the mutant M' and collects the result

as *R'*. Function *computeSusp* computes the suspiciousness score of the mutant based on the formula F (Figure 5.2), and test results R and *R'*. *n2s* keeps the maximum suspiciousness score for each node n. After AlloyFL$_{mu}$ exhausts all mutation operators that are applicable to n, n is added to L if its suspiciousness score $n2s[n]$ is greater than 0. Finally, after all AST nodes are exhausted, L is sorted in descending order of suspiciousness and returned.

***AlloyFL**$_{hy}$.* Inspired by [103], AlloyFL$_{hy}$ assigns the average of suspiciousness scores calculated from both AlloyFL$_{co}$ and AlloyFL$_{mu}$ to each AST node. If a node is not mutable, then AlloyFL$_{hy}$ uses the same suspiciousness score as AlloyFL$_{co}$. The intuition of AlloyFL$_{hy}$ is that AlloyFL$_{mu}$ sometimes perform badly for omission errors in which case AlloyFL$_{co}$ performs relatively well. So AlloyFL$_{hy}$ is designed to combine the strengths of both AlloyFL$_{co}$ and AlloyFL$_{mu}$.

5.3 Distance Metrics

To quantitatively measure how close the ranked nodes are to the real faulty nodes, we follow the spirit of the nearest neighbor distance metric (*NN*) in the program dependence graph (PDG) [111]. Since there is no notion of control dependences in declarative languages like Alloy, we view the Alloy AST as a PDG and adapt the *NN* distance metric on the AST.

The original nearest neighbor distance metric quantifies the percentage of nodes not needing inspection by the programmer using the formula $1 - \frac{|S(R)|}{|G|}$, where $R = \{n_1, n_2, ..., n_k\}$ is the top k returned suspicious nodes

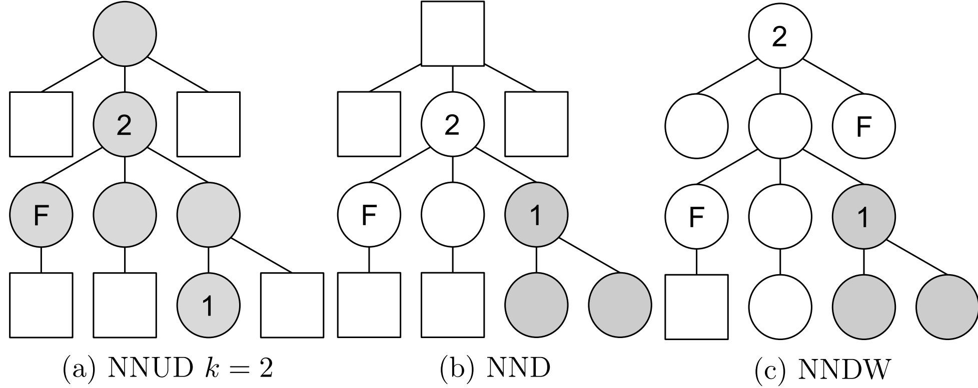

Figure 5.4: Distance Metrics Examples

(n_i, $1 \leq i \leq k$), $S(R)$ is a sphere of all nodes in the graph G such that the maximum distance of any node in S to its closest suspicious node is smaller or equal to the minimum distance of any suspicious node in R to its closest faulty node. Conceptually, the user does a breadth-first search starting with the suspicious nodes, and increasing the distance until a defect is found. The formula computes the percentage of nodes that need not be examined. However, previous studies show that: (1) the percentage of nodes needing inspection is a better estimate than the percentage of nodes not needing inspection [74, 152]; and (2) fault localization techniques should focus on improving absolute rank rather than percentage rank [102]. Therefore, we enhance the *NN* metric to use the absolute number of nodes needing inspection ($|S(R)|$). Techniques which give smaller distance metric values are more accurate. We next describe 3 distance metrics used to evaluate AlloyFL.

Nearest Neighbor Up-Down (*NNUD*). NNUD sets R to the k most suspicious nodes returned. It allows traversing upward (parent) and downward

(children) from the suspicious nodes in the AST until a faulty node is found. Figure 5.4(a) shows the number of nodes one needs to explore from the top two suspicious nodes. The number in the circle represents the position of the node in the ranked list, e.g. 1 means it ranks at the top. "F" shows the faulty node and squares are irrelevant nodes. Circles colored in gray estimate the nodes users need to inspect under NNUD metric with $k = 2$. Since the minimum distance between any of the two suspicious nodes and the faulty node is 1, all nodes that are reachable from the suspicious nodes within a distance of 1 are included. Thus, the metric reports 6, i.e. the size of the gray nodes. NNUD assumes that the programmer may look at the parent or children when inspecting the top k suspicious nodes until a faulty node is found.

Nearest Neighbor Down (**NND*****)***. NND does not allow traversing upward from the suspicious node and it processes suspicious nodes one at a time. Figure 5.4(b) shows how the metric works. From the top 1 suspicious node, we can only traversing downward. Since no faulty node is found, we mark all inspected nodes in gray. Then, NND does a breadth-first search for the second top suspicious node. In this case, a faulty node is found and all descendants within the same distance, i.e. $\leqslant 1$, are included (3 circles colored in white), excluding already visited nodes colored in gray. Finally, NND reports 6, i.e. the size of the inspected nodes in circle. This metric assumes that the users only inspect the children and never reinspect already visited nodes. However, it is possible that the faulty nodes never appear as the descendants of any suspicious node. To avoid this scenario, we append the root node of the entire

AST to the end of the ranked suspicious node list returned by AlloyFL. This makes sure that the metric always terminates with a faulty node found.

Nearest Neighbor Down Worst (**NNDW*****)***. NNDW is similar to NND (only allows traversing downward) except that it assumes the user is unlucky and would inspect all non-faulty nodes before finding the fault. Figure 5.4(c) shows how the metric works. Inspecting the top suspicious node is similar to NND, with the difference occurring when inspecting the second top suspicious node. In this case, we traverse downward and include all non-faulty nodes that have not been visited before (white circles without the faulty node). If a faulty node can be reached from the current suspicious node, then we stop traversing and include all such faulty nodes. In this case, two faulty nodes appear as the children of the second top suspicious node, so we include both faulty nodes. Finally, NNDW returns 10, i.e. all circle nodes. Similar to NND, we append the root node of the entire AST to the end of the suspicious node list.

5.4 Evaluation

We evaluate AlloyFL on 38 real faults collected from Alloy release 4.1, Amalgam [96] and graduate student solutions. These faulty models contain various types of faults, including overconstraints, underconstraints and a mixture of both. In addition, We also extend MuAlloy with the ability to generate higher order mutants [48] and evaluate AlloyFL on 9000 mutant models with exactly two mutant faults. All experiments are performed on Ubuntu 16.04 LTS with 2.4GHz Intel Xeon CPU and 8 GB memory.

In this section, we address the following research questions for both real faults and mutant faults:

- **RQ1.** What is the accuracy and time overhead of AlloyFL?
- **RQ2.** How does the suspiciousness formula affect AlloyFL?

5.4.1 Experiment Setting

Table 5.5 gives an overview for the 18 correct models used to generate mutant faults in the evaluation. Address book (**addr**), Dijkstra mutex algorithm (**dijkstra**), farmer cross-river puzzle (**farmer**), and Halmos handshake problem (**hshake**) are from Alloy's example set. Bad employee (**bempl**), grade book (**grade**), and other groups (**other**) are Alloy translations of access-control specifications used to benchmark Amalgam [96]. Binary tree (**bt**), colored tree (**ctree**), full tree (**fullTree**), n-queens problem (**nqueens**) and singly-linked list (**sll**) are from MuAlloy [142]. Array model (**array**), balanced binary search tree (**bst**), class diagram (**cd**), doubly-linked list (**dll**), finite state machine (**fsm**), and singly-linked list with sorting and counting functions (**stu**) are homework questions we assigned to graduate students.

For each subject, Figure 5.5 shows the number of AST nodes (#**AST**), the number of nonequivalent first-order mutants (**1st**), the number of tests *automatically generated* (**tot**), the number of tests that are expected to be satisfiable (**sat**) and unsatisfiable (**uns**), the number of nonequivalent second-order mutants (**2nd**), and the scope used to run tests or equivalence checks

Model	#AST	1st	#Test			2nd	scp
			tot	sat	uns		
addr	124	62	30	19	11	2.0k	3
array	68	51	36	14	22	1.3k	3
bst	175	167	110	50	60	21.1k	4
bempl	57	35	25	11	14	594	3
bt	61	74	34	20	14	3.4k	3
cd	52	46	24	9	15	1.6k	3
ctree	76	83	22	9	13	4.8k	3
dijkstra	410	183	120	44	76	19.7k	3
dll	92	81	48	22	26	3.6k	3
farmer	180	106	56	33	23	6.5k	4
fsm	85	63	15	3	12	3.0k	3
fullTree	85	100	44	24	20	6.5k	3
grade	77	44	41	23	18	978	3
hshake	136	107	33	10	23	10.3k	4
nqueens	110	101	75	36	39	5.3k	4
other	40	32	21	9	12	558	3
sll	38	31	22	14	8	574	3
stu	201	143	87	40	47	14.2k	3
Sum	2.1k	1.5k	843	390	453	106.0k	

Figure 5.5: Correct Models Information.

(**scp**). Prior works shows that test cases generated by MuAlloy are effective in detecting real faults [133, 142], so we use MuAlloy to generate first-order non-equivalent mutants and the corresponding tests that kill the mutants. We choose to generate second-order mutants for the injected faults because (1) it quickly becomes time consuming to generate mutants with an order higher than 2; and (2) we want to enhance the credibility of our results by using models with more than 1 fault. We filter out second-order mutants that cannot be killed by the generated test suite to make sure at least one fault can be revealed by the test suite. For real faults, we manually inspect all

of the faults and try to fix them without changing the model structure. For example, if the model has a fault in the quantifier body, then we try to fix it without replacing the entire quantifier formula. The expressions/formulas modified due to the fix are labeled as faulty. For mutant faults, following standard practice [66, 67, 157] the mutated nodes are labeled as faulty. We collect 5 real faults from [96], 1 real fault from Alloy release 4.1 and 32 real faults from graduate students. Additionally, we randomly sample 500 second-order mutants for each subject (9000 in total). The models we used to generate mutants contain all correct versions of the real faults.

To evaluate AlloyFL, we use both distance metrics, i.e. NNUD top1 (nnud1), NNUD top5 (nnud5), NNUD top10 (nnud10), NND(nnd) and NNDW(nndw), and the traditional top-k metric, i.e. number of faults in top1, top5 and top10 suspicious nodes. We pick k up to 10 because [62] showed that 98% of practitioners consider a fault localization technique to be useful only if the fault appears in the top-10 suspicious elements. Techniques with smaller values of distance metrics and larger values of top-k metrics are more accurate.

5.4.2 RQ1: AlloyFL Accuracy and Time Overhead

Model	nnud1					nnud5					nnud10					nnd					nndw				
	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy
addr1	78	24	24	**1**	**1**	78	57	48	**5**	**5**	73	57	61	**10**	**10**	50	10	10	**1**	**1**	57	15	15	**1**	**1**
arr1	51	20	20	**14**	**14**	51	56	56	38	**34**	51	61	61	**38**	49	32	61	61	**8**	**8**	41	64	64	**15**	**15**
arr2	29	**1**	**1**	**1**	**1**	26	**2**	**2**	4	5	26	**2**	**2**	4	8	22	**1**	**1**	**1**	**1**	52	**3**	**3**	**3**	**3**
bst1	32	32	32	**5**	**5**	58	55	55	**5**	**5**	65	55	55	**6**	10	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
bst2	3	3	40	4	**1**	15	15	53	14	**5**	28	15	23	**10**	**10**	2	2	64	7	**1**	6	6	76	7	**5**
bst3	19	68	68	**1**	**1**	47	34	32	**5**	**5**	62	34	40	**10**	**10**	6	14	14	**1**	**1**	9	17	17	**1**	**1**
bempl1	19	19	19	**3**	**3**	**7**	19	**7**	10	10	**17**	19	18	**17**	19	6	19	6	**2**	**2**	6	48	6	**2**	**2**
cd1	23	45	45	**4**	**4**	38	45	40	**5**	**5**	45	45	45	**10**	**10**	9	45	15	**3**	**3**	14	58	21	**5**	**5**
cd2	16	34	34	**1**	**1**	25	34	31	**4**	5	25	34	31	**4**	6	6	34	12	**1**	**1**	9	47	16	**2**	**2**
ctree1	**18**	**18**	**18**	39	39	**5**	18	9	56	56	**8**	18	17	21	21	**4**	18	5	61	47	**4**	67	5	71	51
dll1	6	**4**	**4**	**4**	**4**	30	16	16	19	**14**	33	**16**	**16**	19	26	4	**3**	**3**	**3**	**3**	7	**6**	**6**	**6**	**6**
dll2	6	4	4	**1**	**1**	30	16	14	**5**	**5**	36	16	22	**10**	**10**	4	3	3	**1**	**1**	8	7	7	**1**	**1**
dll3	**1**	31	31	2	2	**5**	26	38	**5**	**5**	**7**	26	22	10	10	**1**	11	11	2	2	**1**	13	13	2	2
dll4	6	4	4	**1**	**1**	30	16	16	**4**	5	33	16	16	**4**	8	4	3	3	**1**	**1**	8	7	7	**1**	**1**
farmer1	90	17	17	**9**	**9**	90	94	94	**26**	**26**	79	94	94	**40**	43	54	13	13	**7**	**7**	74	30	30	**12**	**12**
fsm1	50	4	4	**1**	**1**	49	19	15	**5**	**5**	50	19	25	**10**	**10**	71	3	3	**1**	**1**	91	7	7	**1**	**1**
fsm2	59	59	59	**1**	**1**	59	59	59	**5**	**5**	59	59	59	**10**	**10**	69	59	44	**1**	**1**	78	87	52	**1**	**1**
grade1	**1**	19	19	23	4	**5**	19	27	15	**5**	**7**	19	18	25	10	**1**	19	21	42	18	**1**	56	29	71	18
other1	**27**	**27**	**27**	28	28	28	27	28	**5**	**5**	17	27	23	**7**	10	14	27	15	**8**	**8**	18	60	19	**9**	**9**
stu1	7	**4**	**4**	11	11	21	18	18	**8**	9	41	**18**	**18**	24	19	4	**3**	**3**	5	5	12	**11**	**11**	**11**	**11**
stu2	9	**4**	**4**	**4**	**4**	26	18	10	**5**	**5**	47	18	26	**10**	**10**	4	**3**	**3**	4	4	6	11	11	**5**	**5**
stu3	3	**1**	**1**	3	3	9	**4**	5	13	11	24	**4**	10	10	10	2	**1**	**1**	2	2	10	**4**	**4**	10	10
stu4	7	**4**	**4**	11	11	21	18	18	**8**	9	42	**18**	**18**	24	19	4	**3**	**3**	5	5	12	**11**	**11**	**11**	**11**
stu5	7	**4**	**4**	11	11	21	18	18	**8**	9	42	**18**	**18**	24	19	4	**3**	**3**	5	5	12	**11**	**11**	**11**	**11**
stu6	32	10	10	**1**	**1**	38	45	45	**5**	**5**	9	45	45	**7**	10	35	5	5	**1**	**1**	51	7	7	**1**	**1**
stu7	3	**1**	**1**	**1**	**1**	9	**5**	**5**	**5**	**5**	24	**5**	10	10	10	2	**1**	**1**	**1**	**1**	10	**7**	**7**	**7**	**7**
stu8	7	**4**	**4**	**4**	**4**	22	19	19	23	**9**	47	**19**	**19**	23	21	4	**3**	**3**	**3**	**3**	7	**6**	**6**	**6**	**6**
stu9	9	66	66	**1**	**1**	36	66	66	**5**	**5**	47	66	60	**10**	**10**	4	66	86	**1**	**1**	5	166	98	**1**	**1**
stu10	7	**4**	**4**	11	11	22	19	19	**8**	9	47	**19**	**19**	25	**19**	4	**3**	**3**	5	5	12	**11**	**11**	**11**	**11**
stu11	23	**8**	**8**	11	11	41	38	38	**10**	**10**	70	38	38	27	**19**	7	**6**	**6**	9	9	17	16	16	**15**	**15**
stu12	**7**	90	90	11	11	31	29	17	**5**	**5**	46	29	27	**10**	**10**	4	11	11	**3**	**3**	13	20	20	**3**	**3**
stu13	45	116	116	**1**	**1**	63	116	116	**5**	**5**	98	116	102	**10**	**10**	11	116	28	**1**	**1**	18	204	39	**1**	**1**
stu14	34	9	9	**1**	**1**	40	46	46	**5**	**5**	10	46	46	**7**	10	40	5	5	**1**	**1**	52	7	7	**1**	**1**
stu15	7	**4**	**4**	11	11	21	22	22	**8**	9	43	22	22	24	**19**	4	**3**	**3**	5	5	12	**11**	**11**	**11**	**11**
stu16	**1**	32	32	60	**1**	**5**	32	38	56	**5**	**10**	32	25	64	**10**	**1**	32	47	63	**1**	**1**	94	55	94	**1**
stu17	32	**4**	**4**	11	11	38	22	22	11	**9**	**9**	22	22	**9**	19	55	**3**	**3**	5	5	69	**11**	**11**	**11**	**11**
stu18	74	17	17	**4**	**4**	53	**4**	5	5	5	76	**4**	10	10	10	12	**8**	**8**	9	9	14	10	10	**9**	**9**
stu19	3	**1**	**1**	**1**	**1**	12	**5**	**5**	**5**	**5**	24	**9**	10	10	10	2	**1**	**1**	**1**	**1**	4	**1**	**1**	3	3
Avg	22.4	21.5	22.4	8.2	6.1	31.7	30.8	30.8	11.4	9.1	38.9	31.1	31.4	15.9	14.6	14.9	16.5	14.0	7.5	4.7	21.7	32.2	19.6	11.5	7.1
Med	12.5	9.5	13.5	4.0	3.5	29.0	20.5	22.0	5.0	5.0	41.5	20.5	22.5	10.0	10.0	4.0	5.0	5.0	3.0	2.5	12.0	11.0	11.0	5.5	5.0
Std	23.0	26.4	26.4	11.8	7.8	20.2	24.1	24.6	12.6	9.6	22.7	24.2	22.2	11.9	8.9	19.9	23.9	19.7	14.5	7.8	24.4	43.9	21.6	20.4	8.7
Win	6	16	16	23	25	5	5	4	26	26	6	11	7	21	15	5	16	16	23	25	5	13	13	31	33

Figure 5.6: Distance Metrics for Real Faults.

Figure 5.6 shows the distance metric results, i.e. the number of AST nodes to inspect before finding the first fault, of AlloyFL for real faults. We use Ochiai formula for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$. The most accurate AlloyFL techniques w.r.t. each distance metric are highlighted in bold. For each distance metric, we show the results of all AlloyFL techniques per real fault. **Avg**, **Med** and **Std** show the average, median and standard deviation of the corresponding distance metric for each AlloyFL technique over all real faults. **Win** shows the number of times the corresponding AlloyFL technique gives the best distance metric result among all techniques. **Co**, **Un**, **Su**, **Mu** and **Hy** represent AlloyFL$_{co}$, AlloyFL$_{un}$, AlloyFL$_{su}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$, respectively. We can see that AlloyFL$_{hy}$ has the smallest distance metric result in terms of both **Avg** and **Med**, indicating that AlloyFL$_{hy}$ is the most accurate technique in terms of distance metrics for real faults. Moreover, AlloyFL$_{hy}$ is more stable because it has the smallest **Std**. Out of 38 real faults, AlloyFL$_{hy}$ is the most accurate technique in 25 times under NNUD when $k = 1$, 25 times under NND, and 33 times under NNDW. For NNUD when $k = 5, 10$, AlloyFL$_{hy}$ gives the best results in 26 and 15 times, respective, which is close to AlloyFL$_{mu}$. AlloyFL$_{mu}$ is slightly less accurate than AlloyFL$_{hy}$ in terms of **Avg** and **Med**. AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$ perform almost equally bad in terms of **Avg** under NNUD metrics, and AlloyFL$_{un}$ is even worse than AlloyFL$_{co}$ and AlloyFL$_{su}$ under NND and NNDW metrics. All of AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$ are significantly worse than AlloyFL$_{mu}$ and AlloyFL$_{hy}$. AlloyFL$_{co}$ is accurate for omission faults which happen at

the level of paragraph bodies, e.g. when users leave the entire predicate body empty (**stu16**) or miss some conjunct/disjunct constraints at the body of a predicate (**grade1**). On the contrary, AlloyFL$_{mu}$ is not accurate for omission errors because no mutation operator is applicable for an omitted faulty expression/formula. As a consequence, we design AlloyFL$_{hy}$ to leverage the benefits from both AlloyFL$_{co}$ and AlloyFL$_{mu}$. AlloyFL$_{un}$ prioritizes AST nodes that are highlighted the most number of times by the unsat core across all unsatisfiable failing tests and is designed to be comparable or more accurate than using a single unsatisfiable failing test, i.e. the traditional way an Alloy user would debug a faulty model using the unsat core. Our experiments show that the unsat core's accuracy in highlighting suspicious Alloy code is comparable to SBFL (AlloyFL$_{co}$) and significantly worse than MBFL (AlloyFL$_{mu}$ and AlloyFL$_{hy}$).

Figure 5.7 shows the traditional top-k metric results, i.e. the number of top k suspicious nodes that exactly match the faulty nodes, of AlloyFL for real faults. We highlighted the most accurate AlloyFL techniques w.r.t. each top-k metric in bold. **Sum** shows the total number of faults that exactly match the top-k suspicious nodes for each AlloyFL technique over all real faults. **Win** shows the number of times the corresponding AlloyFL technique gives the best top-k metric result among all techniques. Similar to the observation for distance metrics, AlloyFL$_{mu}$ and AlloyFL$_{hy}$ perform equally well and are significantly more accurate than AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$. AlloyFL$_{hy}$ locates 2 more faulty AST nodes than AlloyFL$_{mu}$ for top-1 metric

Model	top1					top5					top10				
	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy
addr1	0	0	0	**1**	**1**	0	0	0	**1**	**1**	0	0	0	**1**	**1**
arr1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
arr2	0	**1**	**1**	**1**	**1**	0	**1**	**1**	**1**	**1**	0	**1**	**1**	**1**	**1**
bst1	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0	**1**	**1**
bst2	0	0	0	0	**1**	0	0	0	0	**1**	0	0	0	**1**	**1**
bst3	0	0	0	**1**	**1**	0	0	0	**2**	1	0	0	0	**5**	1
bempl1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cd1	0	0	0	0	0	0	0	0	**2**	**2**	0	0	0	**3**	2
cd2	0	0	0	**1**	**1**	0	0	0	**1**	**1**	0	0	0	**1**	**1**
ctree1	0	0	0	0	0	**1**	0	0	0	0	**1**	0	0	0	0
dll1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
dll2	0	0	0	**1**	**1**	0	0	0	**2**	1	0	0	0	**2**	**2**
dll3	**1**	0	0	0	0	1	0	0	**2**	**2**	1	0	0	**5**	4
dll4	0	0	0	**1**	**1**	0	0	0	**1**	**1**	0	0	0	**1**	**1**
farmer1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
fsm1	0	0	0	**1**	**1**	0	0	0	**1**	**1**	0	0	0	**1**	**1**
fsm2	0	0	0	**1**	**1**	0	0	0	**1**	**1**	0	0	0	**1**	**1**
grade1	**1**	0	0	0	0	**1**	0	0	0	**1**	**1**	0	0	0	**1**
other1	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0	**2**	**2**
stu1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
stu2	0	0	0	0	0	0	0	0	1	**2**	0	0	0	**2**	**2**
stu3	0	**1**	**1**	0	0	0	**2**	**2**	0	0	0	**2**	**2**	1	1
stu4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
stu5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
stu6	0	0	0	**1**	**1**	0	0	0	**2**	1	1	0	0	**2**	**2**
stu7	0	**1**	**1**	**1**	**1**	0	**2**	**2**	**2**	**2**	0	2	2	**3**	2
stu8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
stu9	0	0	0	**1**	**1**	0	0	0	**1**	**1**	0	0	0	**1**	**1**
stu10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
stu11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
stu12	0	0	0	0	0	0	0	0	**2**	**2**	0	0	0	**2**	**2**
stu13	0	0	0	**1**	**1**	0	0	0	**1**	**1**	0	0	0	**1**	**1**
stu14	0	0	0	**1**	**1**	0	0	0	**2**	1	1	0	0	**2**	1
stu15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
stu16	**1**	0	0	0	**1**	**1**	0	0	0	**1**	**3**	0	0	0	1
stu17	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0
stu18	0	0	0	0	0	0	**1**	**1**	**1**	**1**	0	**1**	**1**	**1**	**1**
stu19	0	**1**	**1**	**1**	**1**	0	**2**	**2**	**2**	**2**	0	2	2	**3**	**3**
Sum	3	4	4	14	16	4	8	8	30	30	9	8	8	44	37
Win	3	4	4	14	16	3	5	5	20	20	4	3	3	23	18

Figure 5.7: Top-k Metrics for Real Faults.

but it locates 7 less faulty AST nodes than AlloyFL$_{mu}$ for top-10 metric. Both AlloyFL$_{hy}$ and AlloyFL$_{mu}$ locate the same number (but different set) of faulty AST nodes in total for top-5 metric. AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$

are comparable to each other and they locate more or less the same number of faulty AST nodes for top-k metrics (except that AlloyFL$_{un}$ and AlloyFL$_{su}$ locate 4 more faulty AST nodes than AlloyFL$_{co}$ for top-5 metric).

Model	nnud1					nnud5					nnud10					nnd					nndw				
	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy
addr	32.3	32.0	32.0	**5.5**	10.0	30.4	31.9	30.5	**7.4**	13.2	27.5	28.6	28.4	**10.4**	12.7	21.1	21.0	19.7	**5.6**	12.5	22.9	38.4	31.7	**8.3**	13.7
array	19.4	7.0	7.5	**2.4**	3.2	18.1	8.1	8.8	**5.7**	6.1	18.8	8.1	9.6	**7.7**	9.6	14.7	6.9	5.9	**2.7**	2.9	21.0	16.4	7.8	5.1	**4.2**
bst	59.8	43.6	42.4	**4.3**	7.5	64.1	50.9	47.9	**6.8**	7.2	71.9	50.7	51.4	**10.8**	10.9	23.5	25.5	19.1	4.4	**3.7**	31.1	51.7	27.7	6.4	**4.3**
bempl	13.0	19.2	17.2	**5.7**	7.7	8.8	16.6	12.6	**6.5**	7.7	11.3	16.6	14.1	**7.0**	9.8	4.7	16.3	10.2	**3.4**	3.9	5.1	46.7	20.8	4.6	**4.6**
bt	23.8	12.9	12.3	**3.1**	3.5	28.3	17.0	17.6	5.8	**5.7**	28.3	17.0	17.8	**8.8**	9.4	12.5	9.4	8.8	2.7	**2.5**	17.5	16.5	12.8	3.8	**3.3**
cd	13.0	13.6	13.5	3.1	**1.8**	16.0	16.4	15.3	5.6	**5.0**	17.1	16.4	16.2	**8.1**	8.4	4.9	10.3	6.1	3.0	**1.5**	6.0	20.9	9.6	4.4	**1.7**
ctree	29.1	17.9	17.8	**5.9**	9.2	24.2	24.6	22.8	**7.3**	11.1	27.4	24.6	24.7	**9.6**	11.9	16.1	13.5	10.7	**4.7**	6.8	19.0	28.3	16.4	**6.3**	7.9
dijkstra	46.5	49.5	53.0	6.6	**5.1**	58.6	46.1	44.0	9.6	**7.1**	71.0	48.9	46.2	13.3	**11.8**	18.0	37.5	23.9	12.0	**6.8**	59.8	127.2	62.2	26.1	**12.6**
dll	22.8	18.5	19.1	4.2	**3.4**	27.7	28.3	26.5	5.8	**5.3**	30.2	28.3	28.6	**8.9**	10.0	10.4	14.2	12.5	3.4	**2.1**	13.0	33.9	19.5	4.8	**2.3**
farmer	45.3	29.2	29.1	**13.1**	15.2	41.7	40.5	39.2	**14.4**	19.4	38.2	40.0	39.2	**16.6**	21.5	27.5	24.0	23.1	12.1	**10.0**	35.1	67.9	49.2	31.3	**12.7**
fsm	27.9	16.1	15.9	11.1	**10.2**	24.2	21.1	20.2	**12.3**	12.9	25.5	22.0	22.4	**13.9**	15.7	21.6	11.8	10.8	10.7	**8.4**	27.3	28.2	16.5	16.8	**10.0**
fullTree	27.5	17.8	17.6	4.1	**3.4**	34.5	23.5	23.6	6.3	**5.5**	34.8	23.5	24.1	**9.6**	9.7	11.2	13.0	10.6	3.6	**2.3**	15.6	26.7	15.9	4.8	**2.8**
grade	12.1	19.3	18.4	2.2	**1.9**	10.7	14.6	12.8	**4.8**	5.3	13.4	14.6	14.3	**6.6**	8.8	5.3	12.9	10.0	2.5	**1.9**	7.6	48.0	31.1	3.3	**2.1**
hshake	38.7	25.8	26.6	**6.1**	7.0	38.2	33.7	32.5	**8.2**	8.3	40.5	35.1	35.1	**11.8**	13.1	24.6	21.7	18.2	6.6	**5.2**	35.8	46.2	26.4	11.2	**6.5**
nqueens	22.5	5.0	5.1	**3.0**	3.5	26.4	7.5	7.6	6.0	**5.7**	26.4	**7.6**	8.5	9.4	10.0	15.8	5.2	3.8	3.8	**3.4**	31.4	15.2	**6.1**	6.9	8.9
other	10.3	10.6	11.3	**2.5**	2.8	8.0	11.3	9.9	**4.9**	5.8	10.3	11.3	11.6	**6.7**	9.9	4.4	10.7	8.2	2.4	**1.8**	4.8	24.0	14.3	3.9	**1.9**
sll	9.1	11.9	11.0	**2.3**	2.5	11.7	13.6	13.3	**4.5**	5.0	11.7	13.6	13.3	**6.2**	7.3	4.9	11.5	9.2	2.1	**2.1**	7.6	22.6	17.2	**2.5**	2.5
stu	45.9	30.8	34.9	**5.2**	6.0	50.1	48.4	43.8	**6.7**	7.6	53.8	49.9	46.5	10.8	**10.4**	15.1	21.9	21.4	4.8	**3.4**	21.1	52.9	32.5	8.1	**3.8**
Avg	27.7	21.2	21.4	5.0	5.8	29.0	25.2	23.8	7.1	8.0	31.0	25.4	25.1	9.8	11.2	14.2	16.0	12.9	5.0	4.5	21.2	39.5	23.2	8.8	5.9
Med	25.6	18.2	17.7	4.2	4.3	27.1	22.3	21.5	6.4	6.6	27.4	22.8	23.2	9.5	10.0	14.9	13.3	10.6	3.7	3.4	20.0	31.1	18.4	5.7	4.2
Std	14.2	11.7	12.3	2.9	3.5	16.1	13.5	12.7	2.5	3.7	18.1	13.8	13.2	2.7	3.1	7.3	7.7	6.1	3.2	3.1	13.7	25.8	14.0	7.8	4.0
Win	0	0	0	12	6	0	0	0	12	6	0	1	0	15	2	0	0	0	4	14	0	0	1	3	14

Figure 5.8: Distance Metrics for Mutant Faults.

Figure 5.8 shows the distance metric results of AlloyFL for mutant faults. We use Ochiai formula for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$. The most accurate techniques w.r.t. each distance metric are highlighted in bold. **Avg**, **Med** and **Std** show the average, median and standard deviation of the corresponding distance metric for each AlloyFL technique over all 9000 mutant faults. Each row shows the distance metric results for various AlloyFL techniques on average over 500 second-order mutant faults. AlloyFL$_{mu}$ is the most accurate technique in terms of **Avg**, **Med** and **Std** under the NNUD metrics ($k = 1, 5, 10$). AlloyFL$_{hy}$ is the most accurate technique in terms of **Avg**, **Med** and **Std** under the NND and NNDW metrics. Both AlloyFL$_{mu}$ and AlloyFL$_{hy}$ significantly outperform AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$. AlloyFL$_{co}$ is the least accurate technique in terms of **Avg** under NNUD metrics ($k = 1, 5, 10$), and AlloyFL$_{un}$ is the least accurate technique under NND and NNDW metrics.

Figure 5.9 shows the traditional top-k metric results of AlloyFL for mutant faults. We highlighted the most accurate AlloyFL techniques w.r.t. each top-k metric in bold. **Sum** shows the sum of the average faults (over 500 mutants) that exactly match the top-k suspicious nodes for each AlloyFL technique over all 18 unique models. **Win** shows the number of times the corresponding AlloyFL technique gives the best top-k metric result among all techniques. Similar to the observation for distance metrics, AlloyFL$_{mu}$ and AlloyFL$_{hy}$ are equally accurate and significantly better than AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$. AlloyFL$_{co}$ is the least accurate technique for top-

Model	top1					top5					top10				
	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy	Co	Un	Su	Mu	Hy
addr	0.1	0.1	0.1	**0.8**	0.6	0.5	0.2	0.3	**1.4**	0.9	0.5	0.2	0.3	**1.5**	1.4
array	0.1	0.6	0.6	0.8	**0.8**	0.4	0.8	0.9	1.2	**1.3**	0.4	0.8	0.9	1.5	**1.6**
bst	0.1	0.2	0.2	**0.7**	0.7	0.1	0.2	0.3	**1.1**	1.0	0.1	0.2	0.3	**1.2**	1.2
bempl	0.2	0.1	0.1	**0.8**	0.6	0.9	0.1	0.6	**1.1**	1.0	1.1	0.1	0.8	1.4	**1.6**
bt	0.1	0.4	0.4	**0.7**	0.7	0.1	0.4	0.4	1.1	**1.1**	0.1	0.4	0.4	1.3	**1.3**
cd	0.2	0.2	0.3	0.8	**0.8**	0.4	0.3	0.4	1.3	**1.3**	0.4	0.3	0.4	1.4	**1.5**
ctree	0.1	0.2	0.2	**0.7**	0.6	0.4	0.2	0.3	**1.0**	0.9	0.4	0.2	0.3	1.1	**1.2**
dijkstra	0.1	0.3	0.3	0.8	**0.8**	0.2	0.4	0.4	**1.2**	1.1	0.2	0.4	0.5	1.3	**1.3**
dll	0.1	0.1	0.2	0.7	**0.7**	0.2	0.1	0.2	**1.3**	1.2	0.3	0.1	0.2	1.4	**1.5**
farmer	0.1	0.1	0.1	**0.6**	0.5	0.3	0.2	0.3	**0.8**	0.7	0.4	0.2	0.3	0.9	**1.0**
fsm	0.1	0.3	0.3	0.4	**0.4**	0.3	0.3	0.4	0.8	**0.8**	0.3	0.3	0.4	0.9	**1.0**
fullTree	0.1	0.3	0.3	0.8	**0.8**	0.2	0.3	0.3	1.1	**1.1**	0.2	0.3	0.3	1.3	**1.3**
grade	0.2	0.1	0.2	**0.9**	0.9	0.7	0.2	0.3	**1.4**	1.4	0.8	0.2	0.4	1.5	**1.7**
hshake	0.1	0.2	0.2	0.7	**0.7**	0.2	0.2	0.3	**1.0**	**1.0**	0.3	0.2	0.3	1.0	**1.1**
nqueens	0.1	**0.8**	**0.8**	0.8	0.7	0.1	1.1	1.1	**1.3**	1.2	0.1	1.1	1.1	1.4	**1.5**
other	0.2	0.2	0.2	**0.9**	0.9	0.9	0.3	0.6	1.2	**1.4**	1.1	0.3	0.8	1.3	**1.7**
sll	0.3	0.0	0.1	**0.8**	0.7	0.5	0.0	0.1	1.4	**1.4**	0.5	0.0	0.1	1.5	**1.5**
stu	0.1	0.1	0.1	**0.8**	0.8	0.2	0.2	0.2	**1.1**	1.1	0.3	0.2	0.2	**1.3**	1.3
Sum	2.1	4.3	4.5	13.2	12.8	6.7	5.6	7.4	20.7	20.2	7.5	5.6	8.1	23.2	24.6
Win	0	1	1	10	7	0	0	0	11	8	0	0	0	3	15

Figure 5.9: Top-k Metrics for Mutant Faults.

1 metric and AlloyFL$_{un}$ is the least accurate technique for top-5 and top-10 metric.

Overall, both AlloyFL$_{mu}$ and AlloyFL$_{hy}$ are significantly more accurate than AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$ for both real faults and mutant faults. AlloyFL$_{su}$ is comparable or more accurate than both AlloyFL$_{co}$ and AlloyFL$_{un}$. AlloyFL$_{co}$ and AlloyFL$_{un}$ are the least accurate techniques and they are comparable to each other. AlloyFL$_{un}$ gives better result for NNUD metrics and worse result for NND and NNDW metrics, compared to AlloyFL$_{co}$. All of AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$ are comparable in terms of top-k

metrics.

MBFL techniques (e.g. AlloyFL$_{mu}$ and AlloyFL$_{hy}$) are the most accurate fault localization techniques and are significantly better than SBFL techniques (e.g. AlloyFL$_{co}$) and SAT-based techniques (e.g. AlloyFL$_{un}$ and AlloyFL$_{su}$). Importantly, our result indicates that AlloyFL$_{mu}$ and AlloyFL$_{hy}$ can more accurately highlight suspicious Alloy code compared to state-of-the-art unsat core.

Because of space limit, we cannot show the time overhead of AlloyFL for individual faults. On average, AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$ take less than 5 sec to run for both mutant faults and real faults. AlloyFL$_{mu}$ takes on average 24.2 sec for mutant faults and 33.7 sec for real faults. AlloyFL$_{hy}$ takes on average 38.0 sec for mutant faults and 67.0 sec for real faults. Both AlloyFL$_{mu}$ and AlloyFL$_{hy}$ run the test for each mutation and AlloyFL$_{hy}$ has the extra overhead to run AlloyFL$_{co}$ and compute the average suspiciousness scores, thus AlloyFL$_{hy}$ is slower than AlloyFL$_{mu}$ and both of them are slower than AlloyFL$_{co}$, AlloyFL$_{un}$ and AlloyFL$_{su}$.

MBFL techniques are significantly slower than SBFL techniques and SAT-based techniques. But since all techniques finish under 2 min on average, the time overhead are reasonable.

5.4.3 RQ2: Suspiciousness Formula Impact

Since AlloyFL$_{un}$ and AlloyFL$_{su}$ do not use suspiciousness formulas, we answer *RQ2* only for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$.

	Formula	nnud1	nnud5	nnud10	nnd	nndw	top1	top5	top10
Co	Tarantula	25.6	32.8	**38.8**	15.0	**20.9**	**0.1**	**0.2**	**0.3**
	Ochiai	**22.4**	**31.7**	38.9	**14.9**	21.7	**0.1**	0.1	0.2
	Op2	28.8	35.7	38.9	23.6	32.1	0.0	0.1	0.2
	Barinel	25.6	32.8	**38.8**	15.0	**20.9**	**0.1**	**0.2**	**0.3**
	DStar	23.4	33.0	38.9	16.5	23.5	**0.1**	0.1	0.2
Mu	Tarantula	10.2	15.2	19.0	8.7	13.3	0.3	0.8	1.0
	Ochiai	8.2	**11.4**	**15.9**	**7.5**	**11.5**	**0.4**	**0.8**	**1.2**
	Op2	12.6	12.8	17.4	10.4	15.4	0.3	0.7	1.0
	Barinel	10.2	15.2	19.0	8.7	13.3	0.3	0.8	1.0
	DStar	**8.2**	**11.4**	16.3	7.8	12.2	0.3	0.7	1.1
Hy	Tarantula	9.9	12.4	**14.5**	5.9	8.1	0.4	**0.8**	1.1
	Ochiai	**6.1**	**9.1**	14.6	**4.7**	**7.1**	**0.4**	0.8	1.0
	Op2	26.2	22.5	19.2	21.8	26.7	0.2	0.3	0.7
	Barinel	10.0	12.4	**14.5**	6.1	8.2	0.3	**0.8**	**1.1**
	DStar	6.6	9.2	15.0	5.1	7.8	0.3	0.7	0.9

Figure 5.10: Formula Impact on AlloyFL for Real Faults.

	Formula	nnud1	nnud5	nnud10	nnd	nndw	top1	top5	top10
Co	Tarantula	29.2	**29.0**	31.0	16.3	23.1	0.1	0.4	0.4
	Ochiai	**27.7**	29.0	**31.0**	**14.2**	**21.2**	0.1	**0.4**	**0.4**
	Op2	29.9	29.5	31.2	15.8	23.0	**0.1**	0.4	0.4
	Barinel	29.2	**29.0**	31.0	16.3	23.1	0.1	0.4	0.4
	DStar	27.8	29.1	31.0	14.4	21.4	0.1	0.4	**0.4**
Mu	Tarantula	10.8	9.8	11.9	8.0	12.3	0.5	1.0	1.1
	Ochiai	**5.0**	**7.1**	**9.8**	**5.0**	**8.8**	**0.7**	**1.2**	**1.3**
	Op2	7.3	8.5	10.6	6.6	11.0	0.7	1.1	1.2
	Barinel	10.8	9.8	11.9	8.0	12.3	0.5	1.0	1.1
	DStar	5.4	7.5	9.9	5.3	9.2	0.7	1.1	1.3
Hy	Tarantula	10.8	9.8	12.8	7.1	8.4	0.5	1.1	1.3
	Ochiai	**5.8**	**8.0**	**11.2**	**4.5**	**5.9**	**0.7**	**1.1**	**1.4**
	Op2	17.0	13.5	12.7	11.8	14.6	0.4	0.9	1.3
	Barinel	11.0	9.7	12.8	7.0	8.2	0.5	1.1	1.3
	DStar	6.4	8.5	11.3	5.2	6.7	0.7	1.1	1.3

Figure 5.11: Formulas Impact on AlloyFL for Mutant Faults.

Figure 5.10 shows the average results for both distance metrics and top-k metrics under different suspiciousness formulas for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$ over 38 real faults. The best results for each metric among all suspiciousness formulas are highlighted in bold. We can see that for AlloyFL$_{co}$, the metric values do not change much for various formulas and Op2 seems to be the worst formula. For AlloyFL$_{mu}$, Ochiai and DStar outperform other formulas and Ochiai is slightly better than DStar. For AlloyFL$_{hy}$, Ochiai seems to be comparable or better than other formulas, followed by DStar. Although Tarantula and Barinel are sometimes the best formulas to use, the improvement is not significant over Ochiai.

Figure 5.11 shows the average results for both distance metrics and top-k metrics under different suspiciousness formulas for AlloyFL$_{co}$, AlloyFL$_{mu}$ and AlloyFL$_{hy}$ over 9000 mutant faults. The best results for each metric among all suspiciousness formulas are highlighted in bold. For AlloyFL$_{co}$, all formulas give similar results. For AlloyFL$_{mu}$ and AlloyFL$_{hy}$, Ochiai gives the best results among all metrics and DStar gives the second best results.

Overall, the choice of formulas does not impact the accuracy of AlloyFL$_{co}$ much for both real faults and mutant faults. Ochiai seems to be the best formula to choose for AlloyFL$_{mu}$ and AlloyFL$_{hy}$ (followed by DStar) for both real faults and mutant faults.

Suspiciousness formulas do not have much impact on the accuracy of SBFL techniques (e.g. AlloyFL$_{co}$). Ochiai formula gives the best result of most metrics for MBFL techniques (e.g. AlloyFL$_{mu}$ and AlloyFL$_{hy}$).

5.5 Summary

This chapter introduces, AlloyFL, a set of 5 fault localization techniques for declarative Alloy models. Our techniques take inspiration from spectrum-based, mutation-based, and SAT-based techniques for imperative code, and build on them to define the core techniques for debugging Alloy. Moreover, we propose 3 new effectiveness metrics to evaluate AlloyFL. We evaluate AlloyFL using a suite of Alloy models including 38 real faults and 9000 mutant faults. The results show that the mutation-based techniques (AlloyFL$_{mu}$ and AlloyFL$_{hy}$) with Ochiai suspiciousness formula perform well and are particularly effective. We believe that our techniques can substantially benefit model developers by speeding up the debugging process, and thus can help improve the quality of software systems.

Chapter 6

Automated model repair for Alloy[3]

In this chapter, we introduce ARepair, a novel generate-and-validate program repair technique for Alloy, which is able to handle Alloy models with multiple faults. ARepair has three main components: (1) AlloyFL, which locates faults at the AST node granularity; (2) RexGen, which systematically generates Alloy expressions (with equivalence pruning rules for relational algebra); and (3) a synthesizer that explores the search space until a model with all passing tests is found. The experimental results show the efficacy of ARepair.

The rest of the chapter is organized as follows. Section 6.1 presents an example. Section 6.2 describes in detail the techniques to repair models. Section 6.3 evaluates ARepair. Section 6.4 summarizes the chapter.

6.1 Example

We use the same example as shown in Figure 5.1. Note that the fault is in the `crossRiver` predicate (highlighted in orange). The predicate enforces eating to happen only after the farmer comes back and not immediately after

[3]Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Automated model repair for alloy. In ASE, 2018.

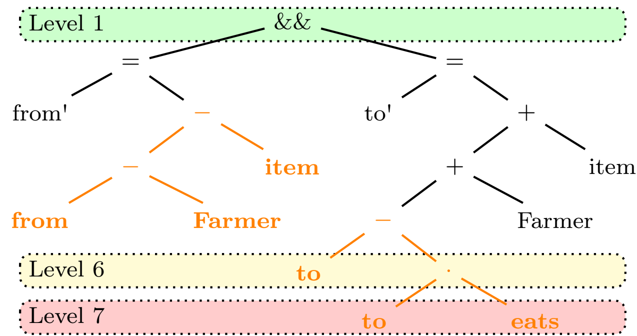

Figure 6.1: First Suspicious Node for Faulty Farmer Example.

the farmer leaves the bank. This modeling error was in Alloy release 4.1 and was fixed in release 4.2. An AUnit test [131] that reveals the fault is shown in Figure 5.1b.

ARepair invokes AlloyFL to locate faults at the AST node granularity. The most suspicious node AlloyFL returns is shown in Figure 6.1. ARepair creates holes to replace each level of AST nodes in a bottom-up fashion. For example, it first creates holes for `to` and `eats` (highlighted in red). Then, ARepair generates a set of candidate expressions for each hole using all signatures/fields/variables in scope, e.g. `Farmer`, `from` and `item`, etc. Next, ARepair enumerates the candidate expressions for each hole and runs all affected tests to see if any test result changes from failing to passing. ARepair keeps the candidate values that make some failing tests pass and preserves the results of passing tests. In this case, ARepair replaces `to` with `none` and now one failing test passes (and no passing test fails). Next, ARepair reruns AlloyFL and finds that the most suspicious node is still the same. In this iteration, ARepair

creates holes for `to` and the relational join operator "." (highlighted in yellow). ARepair keeps synthesizing expressions/formulas under each suspicious node to make failing tests pass. If ARepair cannot make any failing test pass for the suspicious node, then it repeats the same process for the next suspicious node. Note that AlloyFL is a mutation-based technique and it can also repair the model with mutations. Each time AlloyFL is invoked, we check if there is a mutation over the most suspicious node AlloyFL reports that makes some failing tests pass and no passing test fails. If such mutation exists, then we mutate the model and start the next iteration. Finally, if ARepair is able to fix the faulty model, i.e. all tests pass, then it post-processes the fixed model to remove redundant code, e.g. replace "`to - none.eats`" with "`to`", and returns the final model to the user.

Figure 6.2 shows the human written patch (A) and the first patch generated by ARepair (B). We can see that the human written patch fixes the “eating” action both when the farmer crosses the river with (lines 7-10) or without (lines 2-5) an item. The patch ARepair generates deletes the formula that models the farmer’s crossing-river without an item (lines 2-3), and fixes the “eating” action when the farmer crosses the river with an item (lines 6-9). The interesting part is that the patch also changes the domain of variable declaration (lines 4-5), which actually merges both cases when the farmer crosses the river with/without an item. The new domain (line 5) allows the `item` to be the farmer himself and it models the correct semantics corresponding to the deleted formula on lines 2-3. In this case, we validate the equivalence of

(A) A human-written patch.

```
1.   pred crossRiver[from,from',to,to': set Object] {
2.-    (from' = from - Farmer
3.+    (from' = from - Farmer - from'.eats
4.-      && to' = to - to.eats + Farmer ) ||
5.+      && to' = to + Farmer ) ||
6.     (some item: from - Farmer {
7.-      from' = from - Farmer - item
8.+      from' = from - Farmer - item - from'.eats
9.-        && to' = to - to.eats + Farmer + item })}
10.+       && to' = to + Farmer + item })}
```

(B) A patch generated by ARepair.

```
1.   pred crossRiver[from,from',to,to': set Object] {
2.-    (from' = from - Farmer
3.-      && to' = to - to.eats + Farmer ) ||
4.-    (some item: from - Farmer {
5.+    (some item: from + Farmer {
6.-      from' = from - Farmer - item
7.+      from' = from - (Farmer + from'.eats) - item
8.-        && to' = to - to.eats + Farmer + item })}
9.+        && to' = to + Farmer + item })}
```

Figure 6.2: Patches for the faulty farmer model.

generated patch and the human-written patch with a *scope-bounded* analysis using the Alloy analyzer and find that the generated patch is semantically equivalent to the human written patch.

6.2 Technique

In Section 6.2.1, we describe how we create holes and generate expressions to fill in holes (Section 6.2.2). Next, we describe the search strategies (Section 6.2.3). Then, we describe how we run tests without invoking a SAT solver (Section 6.2.4) and the hierarchical caching we use to improve performance (Section 6.2.5). Finally, we describe the enumeration-based repair approach as a whole (Section 6.2.6).

6.2.1 Create Holes

For each suspicious AST node returned by AlloyFL, we create holes at each level of the corresponding AST in a bottom-up fashion. For example, the most suspicious node in the faulty farmer model (Figure 6.1) has 7 levels. We first create holes at level 7 (shown in red) and synthesize new expressions at that level without modifying nodes of other levels. We repeat this process from level 7 to level 1 (root level) until some failing test passes and no passing test fails. The intuition is that AlloyFL is designed to mutate upper-level operator nodes and if the fault cannot be fixed by AlloyFL, then the issue is likely at the lower levels of the AST. This approach also prioritizes patches with smaller perturbations to the original model, which is consistent with the insight – patches that introduce smaller perturbations to the original program are more likely to be correct [16, 69].

Creating a single hole for each node in a given level may not result in valid models. For example, replacing the `&&` node with a hole at level 1 in Figure 6.1 does not make the new program compile. Consequently, the schema to create holes for different AST nodes may vary. ARepair introduces different types of holes, i.e. quantifier holes (denoted by *qh*), logical operator holes (denoted by *loh*), comparison operator holes (denoted by *coh*), implication holes (denoted by *ih*), cardinality holes (denoted by *ch*), boolean holes (denoted by *bh*) and expression holes (denoted by *eh*). The value of *qh* can be one of `all`, `no`, `some`, `lone` or `one`. The value of *loh* can be either `&&` or `||`. The value of *coh* can be one of `=`, `in`, `!=` or `!in`. The value of *ih* can be either `=>` or `<=>`.

Meaning	**Schema**
Cartesian product	$\bar{h}(\phi \times \psi) := eh$
Relational join	$\bar{h}(\phi \bowtie \psi) := eh$
Union	$\bar{h}(\phi \cup \psi) := eh$
Intersection	$\bar{h}(\phi \cap \psi) := eh$
Set difference	$\bar{h}(\phi \setminus \psi) := eh$
Overriding union	$\bar{h}(\phi{+}{+}\psi) := eh$
Domain restriction	$\bar{h}(\phi <: \psi) := eh$
Range restriction	$\bar{h}(\phi :> \psi) := eh$
Transitive closure	$\bar{h}(\hat{}\phi) := eh$
Reflexive transitive closure	$\bar{h}(*\phi) := eh$
Inverse relational join	$\bar{h}(\phi[\psi]) := eh$
Relational transpose	$\bar{h}(\tilde{}\phi) := eh$
Cardinality	$\bar{h}(\#\phi) := eh$
Set comprehension	$\bar{h}(\{\bar{t} : \phi \mid \alpha(\bar{t})\}) := eh$
Identity relation (binary)	$\bar{h}(iden) := eh$
Universe (unary)	$\bar{h}(univ) := eh$
Conjunction	$\bar{h}(\alpha \wedge \beta) := \bar{h}(\alpha)\ loh\ \bar{h}(\beta)$
Disjunction	$\bar{h}(\alpha \vee \beta) := \bar{h}(\alpha)\ loh\ \bar{h}(\beta)$
Implication	$\bar{h}(\alpha \Rightarrow \beta) := \bar{h}(\alpha)\ ih\ \bar{h}(\beta)$
Bi-implication	$\bar{h}(\alpha \Leftrightarrow \beta) := \bar{h}(\alpha)\ ih\ \bar{h}(\beta)$
If-then-else	$\bar{h}(\alpha ? \beta : \gamma) := \bar{h}(\alpha)\ \bar{h}(\beta)\ \bar{h}(\gamma)$
Negation	$\bar{h}(\neg\alpha) := bh\ \bar{h}(\alpha)$
Relational equality	$\bar{h}(\phi = \psi) := \bar{h}(\phi)\ coh\ \bar{h}(\psi)$
Relational containment	$\bar{h}(\phi\ in\ \psi) := \bar{h}(\phi)\ coh\ \bar{h}(\psi)$
Universal quantification	$\bar{h}(\forall \bar{t} : \phi \mid \alpha(\bar{t})) := qh\ \bar{h}(\phi)\ \bar{h}(\alpha(\bar{t}))$
Existential quantification	$\bar{h}(\exists \bar{t} : \phi \mid \alpha(\bar{t})) := qh\ \bar{h}(\phi)\ \bar{h}(\alpha(\bar{t}))$
$\lvert\phi\rvert = 1$	$\bar{h}(one\ \phi) := ch\ \bar{h}(\phi)$
$\lvert\phi\rvert \leq 1$	$\bar{h}(lone\ \phi) := ch\ \bar{h}(\phi)$
$\lvert\phi\rvert \geq 1$	$\bar{h}(some\ \phi) := ch\ \bar{h}(\phi)$
$\lvert\phi\rvert = 0$	$\bar{h}(no\ \phi) := ch\ \bar{h}(\phi)$

Figure 6.3: Hole creation schemas for Alloy Surface Syntax. $\bar{h}(x)$ computes the holes for syntax x. α, β and γ denote formulas which evaluate to true or false. ϕ and ψ denote expressions which evaluate to relations. $\bar{t} : \phi$ denote tuple membership $\bar{t} \in \phi$.

The value of ch can be one of `no`, `lone`, `one` or `some`. The value of bh can be either empty ϵ or `!`. The value of eh can be any expression.

Figure 6.3 shows the meaning of different types of AST nodes and the corresponding schemas to create holes. Each schema is denoted by "$\bar{h}(x) :=$

H", where $\bar{h}(x)$ means the holes created from AST node type x and H shows the way to compute holes. For example, the most suspicious node returned by AlloyFL is a conjunction node (`&&` as shown in Figure 6.1). To create holes for the root node, we can apply the schema for the conjunction node, which states that the holes we should create include a logical operator hole *loh*, holes created from the left child and holes created from the right child. In this case, both the left and right children of the conjunction node are set equality node (`=`), so we recursively apply schemas in Figure 6.3 until no more holes are created. In the end, we would create 4 expression holes, 2 comparison operator holes and a logical operator hole. This step guarantees that if we fill holes with candidate operators/expressions, then the resulting expression/formula is always compilable.

6.2.2 Generating Expressions

The space of candidate fragments for operator holes, e.g. quantifier holes and cardinality holes, are fixed, but the space of candidate fragments for expression holes depends on the expression generator. To generate valid candidate fragments for expression holes, we need to find all atomic expressions in the model that can be used. ARepair has a static analyzer which finds all atomic expressions, i.e. sigs, fields, predicate/function parameters, quantifier variables and let variables, in scope of each expression hole. The holes that share the same set of atomic expressions in scope have the same set of generated candidate fragments.

expression	$e :=$	$uop\ e \mid e\ bop\ e \mid atom_e \mid const$
unary op	$uop :=$	$* \mid \hat{} \mid \tilde{}$
binary op	$bop :=$	$+ \mid \& \mid - \mid \cdot$
atomic expr	$atom_e :=$	$sig_e \mid field_e \mid param_e \mid var_e$
constant	$const :=$	$none \mid iden \mid univ \mid Int \mid 0 \mid 1$

Figure 6.4: Expression generation syntax.

ARepair leverages RexGen to generate expressions following the grammars in Figure 6.4. We enable both static pruning and modulo test pruning in RexGen to prune equivalent expressions. The pruning strategies significantly reduce the number of expressions to consider and make the repair problem tractable.

We describe the modulo test pruning technique with an example. Consider the model shown below:

```
sig Node { link: lone Node } pred p { some n: Node | ?? }
pred t1 { some disj N0: Node | Node=N0 && link=N0->N0 }
pred t2 { some disj N0,N1: Node {
  Node=N0+N1 && link=N0->N1+N1->N0 && p[] } }
```

The model has a signature `Node`, a binary field `link` and a quantifier variable `n`. Implicitly, `n` is of type `Node` and has a cardinality of 1. The model contains two AUnit tests `t1` and `t2`. Suppose we want to generate expressions of type `Node` in the body (denoted by `??`) of the existential quantification, and we can use `n`, `link` and `Node` as the atomic expressions. The following table shows the valuations of four syntactically different expressions, i.e. `n`, `n.link`, `link.(Node-n)` and `(link.Node)&n`, with respect to `t1` and `t2`.

test	n	n	n.link	link.(Node-n)	(link.Node) & n
t1	N0	{N0}	{N0}	∅	{N0}
t2	N0	{N0}	{N1}	{N0}	{N0}
	N1	{N1}	{N0}	{N1}	{N1}

For test `t1`, `n` can only be `N0` and `link` is `N0->N0`. It's easy to see that `n`, `n.link` and `(link.Node)&n` evaluate to the same set {`N0`} and thus are equivalent with respect to `t1`. For test `t2`, `n` can be `N0` or `N1`, and `link` is {`N0->N1, N1->N0`}. `n`, `link.(Node-n)` and `(link.Node)&n` are equivalent with respect to `t2` both when `n=N0` and `n=N1`. So `n` and `(link.Node)&n` are equivalent with respect to the test suite and the modulo test checker can prune either `n` or `(link.Node)&n`. In practice, we keep expressions with smaller sizes, so `(link.Node)&n` will be pruned. If the expression does not contain any free variable, its valuation does not change based on the valuations of free variables. If the expression contains more than one free variable, then we need to enumerate all combinations of possible valuations of the free variables to get the valuations of the expression. If the free variable's cardinality is greater than 1, then its valuation can be any subset of its declaring type. Two expressions are equivalent with respect to a test if their valuations are the same across all combinations of possible valuations of free variables in the scope under test. The expression generator prunes expressions that are equivalent to any existing expression with respect to the entire test suite.

6.2.3 Search Strategies

Given a level of nodes in a suspicious node and the corresponding holes created, ARepair implements two search strategies: *all combinations* and *base*

choice [9].

All combinations. Under this search strategy, ARepair tries all combinations of candidate fragments for all holes until it finds some failing test passed and no passing test fails. This search strategy is typically impractical as the number of holes and the number of candidate fragments for each hole grow. For example, with 4 holes and 100 candidate fragments for each hole, the search space is 10^8. In our implementation, we limit the maximum number of combinations to explore (per level of holes) for this search strategy. Typically, the limit we set is still large, so we stop exploring more combinations *the first time* a combination of candidate fragments makes some failing test pass and no passing test fails. If such a combination is found, we fill the holes with the corresponding fragments and save the changes before starting the next iteration.

Intuitively, we want to first explore combinations of candidate fragments of expression holes with smaller expression sizes, because we assume small-sized expressions are more natural to developers, e.g. `n` vs `(Node-*link.n)`. Figure 6.5 shows how we prioritize exploring combinations of candidate expressions with smaller sizes. Suppose we have n holes ($hole_1$ to $hole_n$) and $hole_i$ has S_i number of candidate fragments. Then we can partition the candidate fragments of $hole_i$ into k_i parts (P_1^i to $P_{k_i}^i$). Next, we create size-n tuples $U = \{(x_1, x_2, \ldots, x_n) \mid \bigwedge_{i=1}^{n} x_i \in [1, k_i]\}$ and sort the tuples first by $\sum_{i=1}^{n} x_i$ and then by $|\max_{\forall i \in [1,n]} x_i - \min_{\forall i \in [1,n]} x_i|$ to gets a ranked list of tuples L. For example, if $n = 2, k_1 = 2, k_2 = 3$, then the ranked tuple list $L = [(1,1),(1,2),(2,1),(2,2),(1,3),(2,3)]$. Finally, we iterate each tuple $(x_1, \ldots, x_n)$

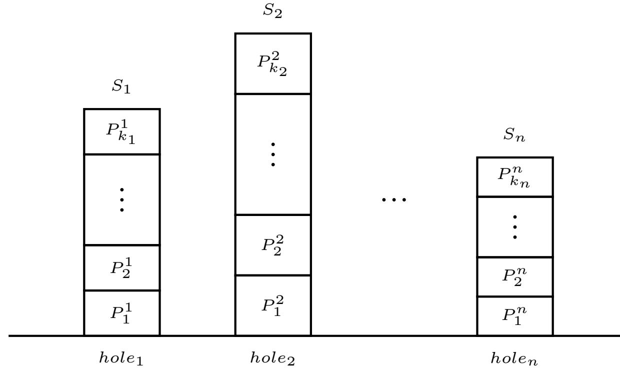

Figure 6.5: All combinations partitions.

in L and explore all combinations of candidate fragments $C = \{(f_1, \ldots, f_n) \mid \bigwedge_{i=1}^{n} f_i \in P^i_{x_i}\}$. Since expressions are generated in a bottom-up fashion, expressions with smaller sizes are generated first, which means expression sizes in P^x_i are smaller than expression sizes in P^x_j if $i < j$. Therefore, the exploration strategy guarantees that combinations of smaller expressions are explored first.

Base Choice. Under this search strategy, ARepair holds candidate fragments of all holes constant except one hole (base choice). It enumerates candidate fragments of $hole_i$ with the candidate fragments of the rest holes unchanged. For each $hole_i$, ARepair explores all candidate fragments and picks the one (f_i) that makes the maximum number of failing tests pass and no passing test fails. Then, ARepair enumerates candidate fragments of $hole_{i+1}$ with the fragment of $hole_i$ set to f_i. ARepair uses this exploration strategy from $hole_1$ to $hole_n$ and saves the final changes as the potential fix. For example, with 4 holes and 100 candidate fragments for each hole, the search space is 400. In practice, the number of generated candidate fragments for an expression hole can be large,

so we set a limit on the number of candidate fragments to explore per hole.

6.2.4 Running Tests

ARepair invokes tests in the expression generation phase (to prune expressions), the fault localization phase (to locate faults) and the repair phase (to validate candidate patches). Since the search space is large and each repair problem contains many tests, invoking all tests at the repair phase takes a majority of the time. Moreover, invoking each test predicate with a SAT solver is expensive. We introduce a technique that determines test satisfiability using Alloy's built-in evaluator (without expensive sat solving) and builds a dependency graph for each test to reduce the number of evaluator calls.

For a given faulty Alloy model, ARepair normalizes the signature multiplicity constraints, the field multiplicity constraints and the signature facts, and creates a formula for each constraint. In the faulty farmer example (Figure 5.1a), the `Object` signature is declared to be an arbitrary set of atoms, so it does not need to be normalized and we create an empty formula (denoted by $\texttt{Object}_{mult}$) which evaluates to true by default. Similarly, the field `eats` is declared to relate an object to a set of objects, so we simply create an empty formula (denoted by $\texttt{eats}_{mult}$). `one sig Farmer` is declared to be a singleton set so we normalize it as `sig Farmer` (remove signature multiplicity constraint) and create a formula `one Farmer` (denoted by $\texttt{Farmer}_{mult}$). Thus, ARepair creates a formula for each signature and field. Since the farmer model does not have any signature facts, we do not need to create any formula for signature

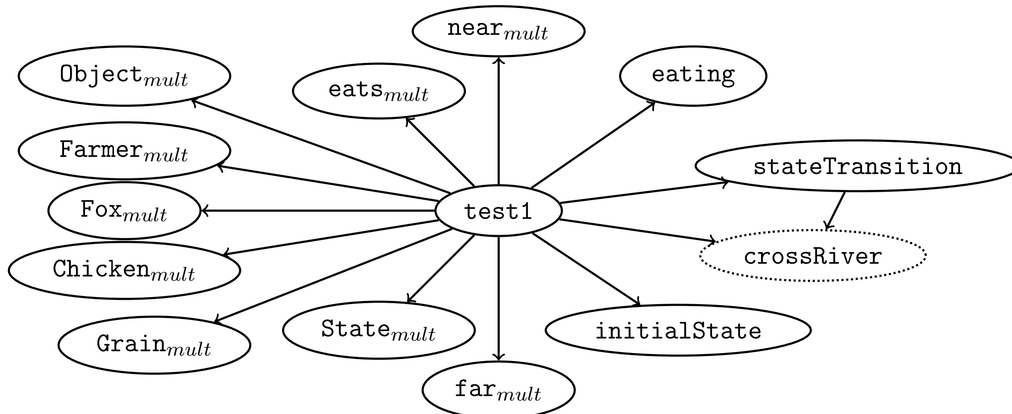

Figure 6.6: Dependency graph for `test1` in Figure 5.1b.

facts. For each fact paragraph, ARepair creates a formula (denoted by the fact name) that is identical to the fact body.

For each AUnit test, we create a dependency graph that encodes the formulas the test depends on. For example, Figure 6.6 shows the dependency graph for `test1` in Figure 5.1b. `test1` depends on all signature/field multiplicity constraints and all fact constraints, because those constraints are enforced by the Alloy analyzer when we invoke the test. Since both `test1` and `stateTransition` directly invoke the `crossRiver` predicate, they both depend on `crossRiver`.

Once we build the dependency graph for each AUnit test, it is easy to compute a test's satisfiability from the formulas the test depends on. Initially, ARepair evaluates each formula the test depends on and stores the satisfiability of each formula. When ARepair enumerates candidate fragments for holes, it only evaluates the affected formulas to determine the satisfiability of the test.

In the faulty farmer example, when ARepair enumerates candidate fragments for holes under the most suspicious AST node (Figure 6.1), the only affected predicate is `crossRiver` and the affected formulas are `stateTransition` and `test1`. To determine the satisfiability of `test1`, we only need to evaluate the body of `stateTransition` and the body of `test1`. Moreover, if any unaffected formula is unsatisfiable, then we know the test is unsatisfiable even without invoking the evaluator. In practice, the technique improves the performance of ARepair because it does not involve any expensive SAT solving and is able to determine the test satisfiability with a minimal number of evaluator calls.

6.2.5 Hierarchical Caching

The evaluator-based approach to determine the test satisfiability can be further improved by our hierarchical caching algorithm. The idea is that we can reuse the previously evaluated result (i.e. valuation) of a formula if its subformulas evaluate to the same set of values as some subformulas we evaluated before. We explain hierarchical caching through the farmer example. Suppose we want to determine the satisfiability of `test1` (Figure 5.1b) by evaluating the fact formula `stateTransition`, and the created holes correspond to nodes at level 7 of the most suspicious AST node, i.e. `to` and `eats` in Figure 6.1 highlighted in red. Also assume that hole $\bar{h}(\mathtt{to})$ is first replaced by fragment `none` and hole $\bar{h}(\mathtt{eats})$ is unchanged. We create a hierarchical cache for `test1` as follows. First, we invoke the evaluator for the fragments of both holes and find that `none` evaluates to $\varnothing$ and `eats` evaluates to {X0→C0, C0→G0}. So we cre-

ate mappings <"none", [∅]> for $\bar{h}$*(to)* and <"eats", [{X0→C0, C0→G0}]> for $\bar{h}$*(eats)*. Since the join operator "·" in level 6 is the lowest common ancestor of both holes in level 7, a mapping <"∅.{X0→C0, C0→G0}", [∅]> is created for the join operator. Note that the key of the join operator is its string representation with all descendant holes replaced by their valuations under `test1`. The value of the mapping is obtained by evaluating the string representation of the join operator, i.e. `none.eats`, which is ∅. We then create a mapping for the body of the declaring `crossRiver` predicate. But because the body has parameters (`from`, `from'`, `to`, `to'`), we need to assign possible values to all parameters and create a mapping for the body <"$A_1A_2...A_n$", [{B_1},{B_2},...,{B_n}]>, where A_i means the string representation of the body (with the join operator "·" replaced with its actual valuation) given ith possible assignment of parameters, and B_i is the corresponding boolean result of the body formula in this case. We finally maps the cached value of `crossRiver`, i.e. [{B_1},{B_2},...,{B_n}], to the satisfiability of the `stateTransition` fact.

If the next fragment of hole $\bar{h}$*(to)* is `item-Object` which evaluates to ∅ ($\bar{h}$*(eats)* is unchanged), then we immediately know that `stateTransition` evaluates to the same result as when hole $\bar{h}$*(to)* is `none`. Because the new keys we computed for other nodes, e.g. the join operator in level 6, are already in the cache. Therefore, we only invoke the evaluator once to evaluate `item-Object` instead of evaluating the big `stateTransition` body to determine its satisfiability. In general, the hierarchical cache reduces the input size of evaluator calls but increase the number of evaluator calls. In practice, we ob-

Algorithm 3: ARepair algorithm.

Input: Faulty Alloy model M, test suite T.
Output: Fully fixed model or partially fixed model.

```
1  canFix = True
2  while canFix do
3      res = runTests(M)
4      if allTestsPassed(res) then return M // Full fix.
5      L = locateFaults(M, res)
6      if isEmpty(L) then return M // Partial fix.
7      canFix = False
8      if isFixed(M, L[0]) then
9          M = applyChange(M, L[0])
10         canFix = True
11     else
12         foreach n ∈ L do
13             patch = synthesize(M, n)
14             if isFixed(M, patch) then
15                 M = applyChange(M, patch)
16                 canFix = True
17                 break
18 return M // Partial fix.
```

serve speed-ups for a majority of repairing problems and few repair problems suffer from a slow-down.

6.2.6 Repair Algorithm

Algorithm 3 shows the algorithm of ARepair. The algorithm takes as input a faulty Alloy model M and a test suite T that reveals the fault. The output is either a fully fixed model if all tests pass or a partially fixed model otherwise. In the worst scenario, ARepair is not able to fix any fault, in which case the partially fixed model is the original faulty model. Initially,

we set *canFix* to *true* (line 1) and enter the loop (line 2). For each iteration in the loop, we first run all tests against M (line 3). If all tests pass, M is returned (line 4). Otherwise, we run AlloyFL to return a ranked list (L) of suspicious AST nodes (line 5). If L is empty, then the algorithm cannot fix the faulty model and it returns the latest state of M (line 6). Otherwise, we set *canFix* to *false* (line 7) and try to fix the faults. The algorithm checks if the most suspicious AST node (*L[0]*) is a potential fix (line 8). The *isFixed* check determines if we want to use the mutation or the synthesizer to fix the model. In general, the *isFixed* method returns true if the mutation makes X failing tests pass and Y passing tests fail, where $X > 0$ and $Y = 0$. In practice, X and Y can be arbitrary numbers as long as $X > Y$ holds, because we want to make sure the algorithm terminates. Since initially we have finite number of failing tests and $X > Y$ makes sure that fewer tests are failing at each iteration. The total number of iterations is bounded by the number of initial failing tests. If *isFixed* (line 8) returns true, then we apply the mutation to M (line 9) and set *canFix* to *true* (line 10). Otherwise, we iterate over the ranked suspicious nodes (line 12) and try to fix the model using the synthesizer. For each suspicious node in L, we invoke the synthesizer to create holes, generate candidate fragments, explore the search spaces and find a potential patch (line 13). Then, the algorithm checks if the patch is a potential fix (line 14). The *isFixed* method in line 14 is similar to the method in line 8. If the patch is a fix, then we apply the patch to the model (line 15) with *canFix* set to *true* and exit the inner loop (line 12-17). Otherwise, we invoke the synthesizer on the

next suspicious node in L. If the synthesizer cannot find a fix after exploring all suspicious nodes, then the algorithm exits the outer loop (line 2) and returns the latest state of M.

If the resulting model passes all tests, ARepair simplifies the model to make it look more natural to the developer. For example, ARepair replaces "`to - none.eats`" with "`to`" because "`none.eats`" always evaluates to an empty set.

6.3 Evaluation

We evaluate ARepair on 38 real faults collected from Alloy release 4.1, Amalgam [96] and Alloy homework solutions from graduate students. These faulty models contain various types of faults, i.e. overconstraints, underconstraints and a mixture of both. We define the number of faults as the number of incorrectly modeled Alloy paragraphs, e.g. signatures, predicates, functions and facts.

We address the following research questions in this section:

- **RQ1.** What is the repair efficacy of ARepair?
- **RQ2.** How does the quality of ARepair generated patches compared to human-written patches?
- **RQ3.** Why is ARepair unable to fix some models?

Model	#AST	#Test	#Flt	all-combinations search strategy								base-choice search strategy							
				#Fix	Status	Types	SS	ES	FL(s)	EG(s)	EE(s)	#Fix	Status	Types	SS	ES	FL(s)	EG(s)	EE(s)
addr1	124	30	1	1	★	M	1	1	41.0	0.0	45.3	1	★	M	1	1	7.3	0.0	8.3
arr1	64	37	1	0	✗	–	1.7e5	1.3e5	36.0	13.5	291.4	1	★	S	4e4	88	1.8	1.9	6.7
arr2	80	37	1	1	★	M+S	3.4e7	5e6	11.6	1.9	3.4e3	1	★	M+S	3.4e7	178	11.4	1.9	16.9
bst1	186	124	1	1	✓	M	1	1	67.6	0.0	74.2	1	✓	M	1	1	46.4	0.0	48.8
bst2	161	124	3	–	∞	–	–	–	–	–	–	0	✗	–	2e17	1.6e5	678.6	597.2	4.8e3
bst3	165	124	2	–	∞	–	–	–	–	–	–	2	✓	M+S	2e8	4e3	241.3	1174	1.5e3
bempl1	51	25	1	0	✗	–	325	325	2.6	0.4	5.3	0	✗	–	325	67	2.6	0.3	5.3
cd1	59	31	2	1	†	M	1e4	993	5.9	1.3	10.6	2	✓	M+S	1.9e5	688	6.0	3.9	12.7
cd2	50	31	1	1	✓	S	966	350	2.3	0.6	6.4	1	✓	S	2.4e5	810	2.2	4.7	11.2
ctree1	71	22	1	1	✓	M	1	1	5.6	0.0	6.5	1	✓	M	1	1	5.5	0.0	6.4
dll1	109	50	2	2	✓	S	4.9e4	8522	19.5	2.3	42.1	2	✓	S	6.7e4	239	19.0	8.3	34.1
dll2	105	50	2	2	✓	M+S	4.9e4	8521	26.7	1.7	46.9	2	✓	M+S	6.6e4	192	27.3	7.7	39.8
dll3	101	50	3	–	∞	–	–	–	–	–	–	0	✗	–	1.7e9	2.1e4	31.3	30.4	94.7
dll4	109	50	1	1	✓	M+S	4.9e4	8384	16.3	1.7	36.2	1	✓	M+S	6.6e4	191	16.5	7.7	28.8
farmer1	180	76	1	–	∞	–	–	–	–	–	–	1	✓	M+S	7.6e13	4556	140.7	1.6e4	4.5e4
fsm1	116	15	2	2	★	M	2	2	7.0	0.0	7.7	2	★	M	2	2	6.9	0.0	7.7
fsm2	93	15	1	1	★	M	1	1	3.8	0.0	4.7	1	★	M	1	1	3.9	0.0	4.7
grade1	71	42	1	–	∞	–	–	–	–	–	–	1	✓	S	1.5e9	1e3	5.2	9.5	739.6
other1	68	22	1	1	★	S	7593	586	2.4	0.7	8.3	0	✗	–	1.7e4	387	2.4	0.7	8.4
stu1	213	98	1	1	✓	S	9.3e4	836	52.6	6.7	85.1	1	✓	S	1.7e6	186	26.3	1.8	46.6
stu2	195	98	2	–	∞	–	–	–	–	–	–	1	✗	S	9.6e12	9905	141.7	40.0	453.5
stu3	237	98	2	–	∞	–	–	–	–	–	–	0	✗	–	1.7e15	3.5e4	347.9	355.9	2.8e3
stu4	190	98	1	1	✓	S	9.3e4	836	53.6	6.6	85.6	1	✓	S	1.7e6	186	26.8	1.9	46.9
stu5	235	98	1	1	✓	S	9.3e4	836	55.0	6.7	87.9	1	✓	S	1.7e6	186	27.5	1.9	48.4
stu6	191	98	3	–	∞	–	–	–	–	–	–	2	✗	M+S	3.8e8	4e3	120.2	22.4	282.4
stu7	174	98	2	2	✓	M+S	5.4e5	1e4	188.9	15.4	265.7	1	✗	S	9.3e10	2.5e4	213.7	291.6	1.7e3
stu8	213	98	1	1	✓	S	1.4e4	1e4	33.4	7.9	78.7	1	✓	S	1.4e4	120	33.4	7.5	54.7
stu9	198	98	1	1	★	M	1	1	49.0	0.0	51.0	1	★	M	1	1	49.9	0.0	51.9
stu10	200	98	1	1	✓	S	9.3e4	836	63.7	6.4	95.9	1	✓	S	1.7e6	186	30.4	1.9	50.5
stu11	221	98	1	1	✓	S	1.4e7	4317	97.6	20.7	174.5	1	✓	S	1.4e7	571	65.3	16.4	131.5
stu12	201	98	2	2	✓	M+S	9.3e4	887	179.8	8.1	224.6	2	✓	M+S	1.7e6	264	152.9	3.5	186.9
stu13	221	98	1	1	★	M	1	1	64.3	0.0	66.4	1	★	M	1	1	64.5	0.0	66.5
stu14	183	98	3	–	∞	–	–	–	–	–	–	2	✗	M+S	3.8e8	4e3	105.5	23.4	266.2
stu15	207	98	1	1	✓	S	9.3e4	836	68.6	6.8	100.7	1	✓	S	1.7e6	186	33.7	2.0	53.6
stu16	113	98	4	–	∞	–	–	–	–	–	–	0	✗	–	5.9e5	6901	43.0	67.9	250.3
stu17	190	98	2	–	∞	–	–	–	–	–	–	1	✗	S	3.5e8	3741	61.0	22.5	205.5
stu18	207	98	3	3	✓	M+S	2.9e9	3.7e5	160.6	6.8	1.1e3	3	✓	M+S	2.9e9	409	115.4	7.4	152.1
stu19	216	98	2	–	∞	–	–	–	–	–	–	1	✗	S	1e13	8221	194.1	9e3	9596

Figure 6.7: ARepair Results. Times are in seconds. – denotes not applicable.

6.3.1 Experiment Setting

Unlike existing datasets that isolate faults for the repair techniques, e.g. Defects4J [55], we use the exact human-written faulty Alloy models as an input to ARepair. We use the Ochiai [2] formula to rank suspicious AST nodes in AlloyFL, because existing studies [130, 156] show that Ochiai is effective. The expression generator generates different sizes of expressions based on the level of the holes in the suspicious AST. We set the expression size to 3 for the deepest level of holes in each suspicious AST. The expression size increases by 1 for holes at depth D_{i-1} compared to holes at depth D_i where $D_i - D_{i-1} = 1$, up to a maximum expression size of 6. For the all-combinations search strategy, we partition the candidate fragments for each hole into 10 parts (i.e. $k_i = 10$ in Figure 6.5), and we set the maximum number of combinations of candidate fragments to explore to 10000 (per level of holes). For the base-choice search strategy, we set the maximum number of candidate fragments to explore for each hole to 1000. The AUnit tests we use to validate the patches are generated so that they are able to detect all non-equivalent mutant models [133]. Additionally, the authors manually inspect the generated tests and add some new tests to cover different corner cases.

We validate the correctness of a generated patch by both inspecting them manually and using the Alloy analyzer to perform a scope bounded equivalence check. The human-written patches are written with the intention of introducing small perturbations that are sufficient to fix the faults. We terminate ARepair once it finds a patch that passes all tests.

All experiments are performed on Ubuntu 16.04 LTS with 2.4GHz Intel Xeon CPU and 16 GB memory. To save space, we denote the all-combinations search strategy as AC and the base-choice search strategy as BC in the following sections.

6.3.2 Repair Efficacy

Figure 6.7 shows the detailed results for ARepair. **Model**, #**AST**, #**Test** and #**Flt** show the name, the number of AST nodes, the number of tests and the number of faults, respectively, for each subject model. **farmer1** is from Alloy release 4.1. **addr1**, **bempl1**, **grade1** and **other1** are from Amalgam [96]. The rest models are from graduate student solutions. Student solutions for the same question share the same test suite. #**Fix** shows the number of faults a search strategy is able to fix. **Status** shows the repair status. ★ means the generated patch is syntactically identical to the human-written patch. ✓ means the generated patch is syntactically different but semantically equivalent to the human-written path. † means the patch is plausible (incorrect but passes all tests). ✗ means ARepair fails to generate a patch that pass all tests. ∞ means the repair times out after 15 hours. **Types** shows whether the fix requires mutations (M) or synthesis (S). **SS** shows the search space size, which is defined as the sum of the number of combinations of candidate fragments (including applied mutations for fixes) to consider in each iteration. **ES** is the actual number of combinations (or mutations) ARepair tried. **FL** is the fault localization time. **EG** is the expression generation time.

EE is the end-to-end time. All times are in seconds.

The entire experiments contain 38 faulty models and 62 individual faults. AC is able to fix 24 models and 31 faults. BC is able to fix 26 models and 42 faults. Additionally, AC times out ($\geq$ 15h) for 12 models while BC finishes all models in 15h. AC is able to fix 2 models that BC is not able to fix, e.g. **other1** and **stu7**. BC is able to fix 5 models that AC is not able to fix (including 1 plausible patch for **cd1**), e.g. **arr1**, **bst3**, **cd1**, **farmer1** and **grade1**. Many models require both mutations and synthesis for a complete fix, e.g. **arr2** and **dll2**. AC's search space ranges from 1 to 2.9e9 and the maximum size of the explored space is 5e6. BC's search space ranges from 1 to 2e17 and the maximum size of the explored space is 1.6e5. We can see that BC explores less of its search space than AC, though BC typically has a much larger search space. In general, BC runs faster than AC, with the exceptions of **cd1**, **cd2** and **stu7**. For AC, the fault localization time ranges from 2.3 sec to 188.9 sec and the maximum expression generation time is 20.7 sec, excluding timeout cases. For BC, the fault localization time ranges from 1.8 sec to 678.6 sec and the maximum expression generation time is 1.6e4 sec. Typically, AC times out for models whose expression generation time is large ($\geq$ 1000s) under BC. A large expression generation time is a reflection of ARepair producing a large number of expressions, resulting in large search spaces. This means that when there are so many combinations of candidate fragments to consider, AC typically times out. In comparison, despite the large number of expressions produced, BC explores much less space and thus is faster. However, BC can

(A) Human-written patch for bst1.

```
1.  pred Sorted() { all n: Node {
2.-   all n2: n.^left | n2.elem < n.elem
3.+   all n2: n.left.*(left+right) | n2.elem < n.elem
4.-   all n2: n.^right | n2.elem > n.elem}}
5.+   all n2: n.right.*(left+right) | n2.elem > n.elem}}
```

(B) ARepair generated patch for bst1.

```
1.  pred Sorted() { all n: Node {
2.-   all n2: n.^left | n2.elem < n.elem
3.+   all n2: n.^left.*right | n2.elem < n.elem
4.-   all n2: n.^right | n2.elem > n.elem}}
5.+   all n2: n.^right.*left | n2.elem > n.elem}}
```

(C) Human-written patch for cd2.

```
1.  pred Acyclic() {
2.-   no c: Class | c = c.ext }
3.+   no c: Class | c in c.^ext }
```

(D) ARepair generated patch for cd2.

```
1.  pred Acyclic() {
2.-   no c: Class | c = c.ext }
3.+   no c: Class | c = c & c.^ext }
```

(E) Human-written patch for stu8.

```
1.  pred Sorted(This: List) {
2.-   all n: Node | n.elem<=n.link.elem }
3.+   all n: Node | some n.link => n.elem<=n.link.elem }
```

(F) ARepair generated patch for stu8.

```
1.  pred Sorted(This: List) {
2.-   all n: Node | n.elem<=n.link.elem }
3.+   all n: link.Node | n.elem<=n.link.elem }
```

Figure 6.8: Comparison of ARepair generated patches and human-written patches.

run into its local optimum. For example, the explored space for **other1** is less than 600 for both BC and AC, but BC cannot fix the model. Overall, AC and BC are complementary and BC is superior in the sense that it takes less time to run and fixes more faults.

6.3.3 Patch Quality

To answer **RQ2**, we find that BC generates 26 patches that pass all tests (all patches are correct and 7 patches exactly match human-written patches).

AC generates 24 patches that pass all tests (23 patches are correct; 7 patches exactly match human-written patches; and 1 patch is plausible but incorrect). We compare the generated patches that are syntactically different but semantically equivalent to human-written patches. In addition to patches for the faulty farmer model (Figure 6.2), Figure 6.8 compares ARepair generated patches and human-written patches for **bst1** (A and B), **cd2** (C and D) and **stu8** (E and F). The `Sorted` predicate in **bst1** models that the value of the current node should be greater than values of its left descendants and less than values of its right descendants. The developer incorrectly use "`n.^left`" to represent the domain of `n`'s left descendants. The correct domain should be "`n.left.*(left+right)`" as shown in the human-written patch. The generated patch restricts the domain to be "`n.^left.*right`" which means all nodes that can be reachable from `n` by first following one or more `left` relation and then zero or more `right` relation. The `Acyclic` predicate in **cd2** models that a class does not transitively extend itself. The faulty model does not consider the transitivity requirement, which is fixed in the human-written patch by replacing "`c = c.ext`" with "`c in c.^ext`". The generated patch uses "`c = c & c.^ext`" which states that no class is equal to the intersection of the class and all its subclasses, transitively. The `Sorted` predicate in **stu8** models a linked list sorted in descending order of the node values. The faulty model does not allow the existence of any list with a single node (without any `link`). The human-written patch allows such cases by stating that if a node `n` has a subsequent node following the link, then its value should be less than or equal

to the value of its subsequent node. The generated patch instead modifies the domain to restrict the less than or equal relation only applying to nodes that have a subsequent node.

The authors check the correct patches that are syntactically different from human-written patches and find that these patches are easy to understand in general. There are rare cases that ARepair generates some complex expressions that can be further simplified through semantic reasoning. Additionally, ARepair generates a patch which fixes a fact instead of the predicate the developer would fix for **ctree1**.

6.3.4 Limitation

To answer **RQ3**, we manually inspect all faulty models that ARepair is unable to fix. The reasons are categorized as follow:

1. The repair requires synthesizing predicate and function calls. For example, one of the property to fix in **bst2** requires invoking predicates and functions.

2. The repair requires moving a field declaration from one signature to another, e.g. **bempl1**.

3. The repair requires creation of new syntactic structures. For example, **dll3** models a property using a single quantifier, but the model needs two. **stu2** has a formula with the structure $(\alpha \Rightarrow \beta) || \gamma$, but the correct fix requires $\alpha \Rightarrow \beta \, else \, \gamma$, where α, β and γ are formulas. **stu6** is overconstrained and the fix requires creating a disjunction of a formula and an existing formula.

dll3 and **stu16** have empty predicate and require ARepair to synthesize formulas from scratch.

4. Both AC and BC search strategies are greedy and may run into a local optimum. For example, a correct patch of **other1** requires changing two formulas at the same time and BC runs into a local optimum that leads to a repair failure. Similarly, AC runs into a local optimum for **arr1**.

We find that the majority of the faults ARepair is unable to fix fall under category 3, followed by category 4. To handle faults in category 3, we can add repair templates that introduce new syntactic structures if the current version of ARepair is not able to find a correct patch. New search strategies can be designed to address faults under category 4. From our experiment, ARepair is able to handle majority of the faulty models (28 out of 38) and we plan to handle the limitations in future works.

6.4 Summary

This chapter introduces a *generate-and-validate* repair technique, ARepair, to fix faulty Alloy models. ARepair leverages AlloyFL, RexGen and a synthesizer to repair various kinds of faults. ARepair is enumeration-based and it enbodies two search strategies, i.e. the all-combination strategy and the base-choice strategy. ARepair implements various optimizations, including the use of modulo test input pruning to remove equivalent expressions, the construction of dependency graph to reduce evaluator calls, and the employ-

ment of a hierarchical cache to reduce evaluator input size. The experimental results show that ARepair works well in fixing real faulty models.

Chapter 7

Related Work

This chapter presents an overview of the work related to the contributions of this dissertation. There has been a lot of work on expression generation [63], program sketching [46], automated program fault localization [51] and automated program repair [149]. These work mainly focuses on imperative languages like C and Java. We bring all of the above technologies to the declarative language Alloy, which is a first-order relational logic with transitive closure. The fundamental differences between imperative languages and declarative languages enable a set of new techniques to solve existing problems. This chapter is organized as follows. Section 7.1 discusses work related to expression generation. Section 7.2 discusses work related to program sketching. Section 7.3 discusses work related to fault localization. Section 7.4 discusses work related to program repair. Section 7.5 discusses work related to Alloy.

7.1 Expression Generation

Expression generation serves as the fundamental step for program synthesis. Enumeration algorithms include bottom-up enumeration [8, 139], used by RexGen, and top-down enumeration [24]. EuSolver [8] has been one of the

most prominent solvers in Syntax-Guided Synthesis (SyGuS) competitions. FlashMeta [108] uses version-space algebra to concisely represent a large number of programs. Neither EuSolver nor FlashMeta focus on relational expressions, which can generate a large number of equivalent expressions. Our work proposes a number of pruning rules that substantially reduce the number of equivalent expressions, thus providing basis for practical synthesis with relational expressions.

Search space pruning of expression generation is important because search spaces for any realistic programming language quickly become intractable. Pruning techniques include indistinguishability of expressions modulo a set of inputs [8, 139] and partial evaluation of incomplete expressions [24]. Knowledge about operator properties has also been used to explore equivalent expressions, either after expression generation [106] or by applying an automated transformation to the grammar which represents candidate programs [63]. However, most techniques have only been explored in the domains of integers, booleans, and abstract data types, all of which have less comprehensive sets of equivalence rules than our work with the domains of sets and relations.

Applications of expression generation are quite common. For example, program synthesis has attracted attention for a few decades [83], and researchers have applied it in a variety of domains [23, 24, 27, 36, 61, 82]. Program sketching [129] is another example, which demonstrated the opportunities to apply modern solver technology to the synthesis problem, and introduced the counter-example guided inductive synthesis paradigm to pro-

gram synthesis. Sketch requires the user to provide generators of expressions for expression holes [7, 38, 46, 125]. While most work on sketching is in the context of synthesis, SketchFix [37] applies sketching to the problem of program repair [31, 50, 78, 113, 150], i.e., correcting faulty lines of code. Synthesis from examples, the inspiration behind test valuations, has also been extensively studied [6, 104]. Notably, synthesis from examples has been successfully employed in commercial products [34]. The key enabler of all of the above applications is efficient expression generation; RexGen is the first work that addresses generation for relational algebra.

7.2 Program Sketching

Program sketching [7, 46, 123, 125–129] is a form of program synthesis, which is a mature yet active research topic [11, 23, 25, 27, 36, 61, 64, 82, 100, 123]. Researchers have proposed program synthesis techniques for a number of languages, including synthesis of logic programs, e.g., using inductive synthesis based on positive and negative examples [20]. However, prior work has not addressed the complexity of synthesis in the presence of quantifiers, transitive closure, relational operators, and more generally, formulas that express structurally complex properties, which are the focus of our work.

The Sketch system [126] takes as input a partial program in the Java-like Sketch language, and uses SAT and inductive synthesis in a counterexample-guided loop. Sketch requires users to provide generators for expression fragments for expression holes. The JSketch tool translates Java to Sketch to allow

sketching Java programs [46]. Some tools focus on specific kinds of programs to sketch, such as PSketch for concurrent data structures [128].

SyPet [23] introduced a novel use of Petri nets in synthesizing sequences of method invocations for complex APIs using tests. EdSketch [38] and EdSynth [158] introduced an optimized backtracking search for completing Java sketches using test executions for pruning. Test-Driven Synthesis iteratively builds a C# program such that it satisfies all tests [106]. Component-based synthesis builds programs by combining components from given libraries, e.g., work in this line used I/O oracles to synthesize loop-free programs [47].

ASketch shares the spirit of storyboard programming, which uses user-provided graphical representations of data structures to synthesize imperative code that performs desired data structure manipulations based on the insight that it can be easier and more intuitive for a user to provide concrete data structure manipulations than to write the code [125]. Our test valuations make use of a similar insight.

7.3 Fault Localization

Automated debugging of Alloy models can be traced back to Alloy's early days when highlighting unsat cores in unsatisfiable Alloy formulas was introduced [120]. Moreover, for satisfiable formulas, Alloy's symmetry breaking indirectly supports debugging by allowing the user to inspect fewer instances [29, 58, 97, 121]. More recent work on Amalgam allows the user to ask questions of the form "why a tuple is or is not in a relation" for a chosen

instance [96]. While Amalgam provides a useful tool to aid debugging by allowing the user to enhance their understanding of the model by asking a series of questions, the restricted form of the questions limits its effectiveness, e.g., the user cannot ask why certain formulas hold or not, or why certain relations are empty.

A number of approaches assist users to write correct Alloy models. Montaghami and Rayside [90, 91] enable Alloy users to more easily provide partial instances, which are tangible, expressive example solutions that aid in writing correct, complete models. Sullivan et al. [134] follow the spirit of JUnit and introduce a test automation framework for Alloy by defining test case, test execution and model coverage. AUnit has enabled further test automation efforts for Alloy, ranging from automated test generation to mutation testing [133, 142].

While our focus in this paper is on declarative models written in Alloy, fault localization for imperative languages is a well-studied area. AlloyFL implements spectrum-based, mutation-based, and SAT-based techniques. Among these, spectrum-based techniques [1, 2, 15, 21, 51, 52, 72, 103, 112], are the most widely studied; they focus on collecting execution information, such as statements and methods. Mutation-based fault localization techniques [92, 101] were introduced more recently. They perform mutations on the faulty program to study their impact on the test execution results and determine likely faulty locations. SAT-based techniques use either the minimal satisfiability [33] or the negation of maximal satisfiability [53] to identify suspicious code.

A number of other techniques have also been proposed for fault localization. Comparing program states between passing and failing tests has shown to be highly effective and was pioneered by delta debugging [161, 162], which has led to various other approaches [13, 35, 163]. Statistic-based approaches [75, 152] focus on determining the likelihood of different portions of a program being faulty. Feedback-based debugging [71, 73] is an interactive fault localization approach that utilizes execution traces and user feedback. Program slicing [4, 5, 80] isolates relevant program elements that can trigger the execution traces that lead to errors.

7.4 Program Repair

The *generate-and-validate* repair techniques apply a set of code transformations to generate program candidates and validate each candidate under the given test suite. These techniques implement different search strategies, e.g. genetic algorithms [149], semantic search [57], random search [109] and adaptive search [148], to explore the immense search space of repair candidates. Researchers also proposed other repair techniques that remove program functionalities [110], create program variants [12, 17], leverage dynamic program state [39, 40, 158], or focus on improving performance by removing bottlenecks in concurrent programs [159]. *Astor* [87] is a repair library that implements existing techniques to fix Java code. Techniques that prioritize patches are built based on human-written code [59, 77, 122, 154], historical data [22, 70], document analysis [76, 116, 155], anti-patterns [135] and test generation [153].

The *constraint-solving* repair techniques use the semantics of the faulty program and translate the repair problem into a constraint solving problem. Then, the constraint solving problem is solved by an off-the-shelf solver to find a repair that satisfies all inferred specifications. The constraints can be inferred from test executions [18, 79, 118] or semantic analysis [16, 56, 69, 99]. Other techniques use formal specifications [32, 60, 124, 147] or infer invariants [19, 54, 107, 117] to fix programs.

The fundamental idea of declarative debugging is that the programmer (or some oracle) has an intended interpretation of the program and debuggers can query the programmer to obtain this information. The debugger compares the intended interpretation of a (buggy) program with its (incorrect) actual behavior on some computation. The cause of the difference is isolated to a small section of code which must contain a bug. Declarative debugging was first introduced in Prolog [119] and then extended for functional and logic programs [93, 94, 105]. Researchers also developed program repair technique for SQL [10, 30].

7.5 Alloy

Alloy is a well studied lightweight modeling approach that has been applied in various domains, including software design [84, 85], networking [114], and security [81, 98] Various approaches assist Alloy users to build their models correctly, e.g., by improving scenario exploration [96, 97], supporting state modeling [26, 44, 45, 86, 132], highlighting unsat cores [120, 137, 138], and cre-

ating tests [133, 142]. Over the past years, many extensions have been built for Alloy. Alloy* [88] allows users to write models in second order logic. Electrum [14] is used to validate the train system. Molina et al. [89] uses genetic programming algorithm to generate Alloy specification..

Chapter 8

Conclusion and Future Work

Since writing models correctly is challenging, models often have faults. We propose to alleviate the manual effort put into debugging models by automating the expression generation, fault localization and repairing. Specifically, we implement our ideas in a widely used declarative language, i.e. Alloy. We first introduce RexGen, a expression generator for Alloy, which systematically generates non-equivalent Alloy expressions. Then, we introduce ASketch a solver-based sketching framework. ASketch takes as input a partial Alloy model, a generator and a set of AUnit tests that capture the desire properties of the model. ASketch is able to complete the partial Alloy model such that all tests pass. We next introduce AlloyFL, a set of fault localization techniques for Alloy, which takes as input a faulty Alloy model and a suite of AUnit tests. AlloyFL returns a list of Alloy AST nodes in the descending order of suspiciousness. Finally, we propose ARepair, an automated repair technique that leverages both RexGen and AlloyFL. It takes as input a faulty Alloy model and a suite of AUnit tests and outputs a fixed Alloy model that passes all tests. We believe our work in this thesis can reduce the manual debugging efforts of Alloy users and help them write correct specification.

We plan to collect more real-world faulty Alloy models to better understand the reasons and types of faults developers make. This can further help us design better debugging/repair tools to reduce human effort involved in debugging Alloy models. It would be useful to modify the official Alloy analyzer to record developer activities when they write and fix models. A user study of AlloyFL and ARepair would be good to check if our tool can actually help Alloy developers debug/fix their models faster.

Bibliography

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *JSS*, 2009.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION*, 2007.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *ASE*, 2009.

[4] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE*, 1995.

[5] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI*, 1990.

[6] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In Natasha Sharygina and Helmut Veith, editors, *CAV*, 2013.

[7] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[8] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS*, 2017.

[9] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2008.

[10] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Specifying and querying database repairs using logic programs with exceptions. In *Flexible Query Answering Systems*. 2001.

[11] Rastislav Bodík and Barbara Jobstmann. Algorithmic program synthesis: Introduction. *STTT*, 2013.

[12] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE, 2013.

[13] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE*, 2005.

[14] Alcino Cunha and Nuno Macedo. Validating the hybrid ertms/etcs level 3 concept with electrum. In *ABZ*, 2018.

[15] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *ECOOP*, 2005.

[16] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *CAV*, 2016.

[17] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, 2010.

[18] Favio Demarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA*, 2014.

[19] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, 2006.

[20] Yves Deville and Kung-Kiu Lau. Logic program synthesis. *The Journal of Logic Programming*, 1994.

[21] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, 1999.

[22] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *POPL*, 2016.

[23] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex APIs. In *POPL*, 2017.

[24] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[25] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[26] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. DynAlloy: Upgrading Alloy with actions. In *ICSE*, 2005.

[27] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and interactive synthesis of code snippets. In *ICSE*, 2014.

[28] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *TSE*, 2013.

[29] Juan Pablo Galeotti, Nicolas Rosner, Carlos G. Lopez Pombo, and Marcelo F. Frias. Taco: Efficient sat-based bounded verification using symmetry breaking and tight bounds. *TSE*, 2013.

[30] Divya Gopinath, Sarfraz Khurshid, Diptikalyan Saha, and Satish Chandra. Data-guided repair of selection statements. In *ICSE*, 2014.

[31] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *TACAS*, 2011.

[32] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *TACAS*, 2011.

[33] Divya Gopinath, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *ASE*, 2012.

[34] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *CACM*, 2015.

[35] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE*, 2005.

[36] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *CAV*, 2011.

[37] Jinru Hua and Sarfraz Khurshid. A sketching-based approach for debugging using test cases. In *ATVA*, 2016.

[38] Jinru Hua and Sarfraz Khurshid. EdSketch: Execution-driven sketching for Java. In *SPIN*, 2017.

[39] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Sketchfix: A tool for automated program repair approach using lazy candidate generation. In *FSE*, 2018.

[40] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *ICSE*, 2018.

[41] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 1901.

[42] Daniel Jackson. Alloy: A lightweight object modelling notation. *TSE*, 2002.

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

[44] Daniel Jackson and Alan Fekete. Lightweight analysis of object interactions. In *TACS*, 2001.

[45] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ISSTA*, 2000.

[46] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. JSketch: Sketching for Java. In *FSE*, 2015.

[47] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[48] Yue Jia and Mark Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 2009.

[49] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *TSE*, 2011.

[50] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, 2005.

[51] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.

[52] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[53] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *PLDI*, 2011.

[54] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. Vejovis: suggesting fixes for javascript faults. In *ICSE*, 2014.

[55] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, 2014.

[56] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. Minthint: automated synthesis of repair hints. In *ICSE*, 2014.

[57] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *ASE*, 2015.

[58] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *SAT*, 2003.

[59] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *ICSE*, 2013.

[60] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *CAV*, 2015.

[61] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.

[62] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *ISSTA*, 2016.

[63] Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. An update on deductive synthesis and repair in the leon tool. In *SYNT Workshop*, 2016.

[64] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, 2010.

[65] E. Larson and A. Kirk. Generating evil test strings for regular expressions. In *ICST*, 2016.

[66] Tien-Duy B. Le, David Lo, and Ferdian Thung. Should i follow this fault localization tool's output? *ESE*, 2015.

[67] Tien-Duy B. Le, Ferdian Thung, and David Lo. Theory and practice, do they match? A case with spectrum-based fault localization. In *ICSME*, 2013.

[68] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.

[69] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *FSE*, 2017.

[70] Xuan-Bach D. Le, David Lo, and Claire Le Goues. History driven program repair. In *SANER*, 2016.

[71] Xiangyu Li, Shaowei Zhu, Marcelo d'Amorim, and Alessandro Orso. Enlightened debugging. In *ICSE*, 2018.

[72] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[73] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. Feedback-based debugging. In *ICSE*, 2017.

[74] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *TSE*, 2006.

[75] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: Statistical model-based bug localization. In *FSE*, 2005.

[76] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. R2fix: Automatically generating bug fixes from bug reports. In *ICST*, 2013.

[77] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *FSE*, 2017.

[78] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *FSE*, 2015.

[79] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *FSE*, 2015.

[80] James R. Lyle and Mark Weiser. Automatic Program Bug Location by Program Slicing. In *ICCA*, 1987.

[81] Ferney A. Maldonado-Lopez, Jaime Chavarriaga, and Yezid Donoso. Detecting network policy conflicts using alloy. In *ABZ*, 2014.

[82] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. *PLDI*, 2005.

[83] Zohar Manna and Richard Waldinger. Toward automatic program synthesis. *CACM*, 1971.

[84] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class diagrams analysis using Alloy revisited. In *MODELS*, 2011.

[85] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic differencing for class diagrams. In *ECOOP*, 2011.

[86] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, 2001.

[87] Matias Martinez and Martin Monperrus. ASTOR: a program repair library for java (demo). In *ISSTA*, 2016.

[88] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *ICSE*, 2015.

[89] Facundo Molina, César Cornejo, Renzo Degiovanni, Germán Regis, Pablo F Castro, Nazareno Aguirre, and Marcelo F Frias. An evolutionary approach to translate operational specifications into declarative specifications. In *BSFM*, 2016.

[90] Vajih Montaghami and Derek Rayside. Extending alloy with partial instances. In *ABZ*, 2012.

[91] Vajih Montaghami and Derek Rayside. Staged evaluation of partial instances in a relational model finder. In *ABZ*, 2014.

[92] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, 2014.

[93] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997.

[94] Lee Naish. A three-valued declarative debugging scheme. In *Computer Science Conference*, 2000.

[95] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *TSE*, 2011.

[96] Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. The power of "why" and "why not": Enriching scenario exploration with provenance. In *FSE*, 2017.

[97] Tim Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Aluminum: Principled scenario exploration through minimality. In *ICSE*, 2013.

[98] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave tool for firewall analysis. In *LISA*, 2010.

[99] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *ICSE*, 2013.

[100] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

[101] Mike Papadakis and Yves Le Traon. Metallaxis-fl: Mutation-based fault localization. *STVR*, 2015.

[102] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.

[103] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *ICSE*, 2017.

[104] Yu Pei, Carlo A. Furia, Martín Nordio, and Bertrand Meyer. Automated program repair in an integrated development environment. In *ICSE*, 2015.

[105] Luís Moniz Pereira. Rational debugging in logic programming. In *ICLP*, 1986.

[106] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. *PLDI*, 2014.

[107] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[108] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A framework for inductive program synthesis. In *OOPSLA*, 2015.

[109] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[110] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, 2015.

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

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

[113] Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In *FM*, 2016.

[114] Natali Ruchansky and Davide Proserpio. A (not) NICE way to verify the Openflow switch specification: Formal modelling of the Openflow switch using Alloy. *SIGCOMM*, 2013.

[115] Salman Saghafi, Ryan Danas, and Daniel J. Dougherty. *Exploring Theories with a Model-Finding Assistant*. 2015.

[116] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: effective object oriented program repair. In *ASE*, 2017.

[117] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE*, 2012.

[118] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, 2016.

[119] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, 1983.

[120] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *ASE*, 2003.

[121] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Appl. Math.*, 2007.

[122] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *PLDI*, 2015.

[123] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *CAV*, 2015.

[124] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, 2013.

[125] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *FSE*, 2011.

[126] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[127] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. *PLDI*, 2007.

[128] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *PLDI*, 2008.

[129] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[130] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*, 2013.

[131] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. AUnit: A Test Automation Tool for Alloy. In *ICST*, 2018.

[132] Allison Sullivan, Kaiyuan Wang, Sarfraz Khurshid, and Darko Marinov. Evaluating state modeling techniques in alloy. In *SQAMIA*, 2017.

[133] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Automated test generation and mutation testing for Alloy. In *ICST*, 2017.

[134] Allison Sullivan, Razieh Nokhbeh Zaeem, Sarfraz Khurshid, and Darko Marinov. Towards a test automation framework for Alloy. In *SPIN*, 2014.

[135] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *FSE*, 2016.

[136] Alloy Team. `http://alloy.mit.edu/alloy/documentation/alloy4-grammar.txt`.

[137] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM*, 2008.

[138] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.

[139] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. In *PLDI*, 2013.

[140] Kaiyuan Wang. muAlloy – an automated mutation system for Alloy. Master's thesis, University of Texas at Austin, May 2015.

[141] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Automated model repair for alloy. In *ASE*, 2018.

[142] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. MuAlloy: A Mutation Testing Framework for Alloy. In *ICSE*, 2018.

[143] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. Systematic generation of non-equivalent expressions for relational algebra. In *ABZ*, 2018.

[144] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. Asketch: A sketching framework for alloy. In *FSE*, 2018.

[145] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. Fault localization for declarative models in Alloy. In *eprint arXiv:1807.08707*, 2018.

[146] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. Solver-based sketching Alloy models using test valuations. In *ABZ*, 2018.

[147] Yi Wei, Yu Pei, Carlo A. Furia, Lucas Serpa Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *ISSTA*, 2010.

[148] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*, 2013.

[149] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[150] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[151] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 2014.

[152] W. Eric Wong, Vidroha Debroy, and Dianxiang Xu. Towards better fault localization: A crosstab-based statistical approach. In *IEEE Transactions on SMC*, 2012.

[153] Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*, 2017.

[154] Qi Xin and Steven P Reiss. Leveraging syntax-related code for automated program repair. In *ASE*, 2017.

[155] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *ICSE*, 2017.

[156] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *ICSME*, 2014.

[157] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *FSE*, 2014.

[158] Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. Edsynth: Synthesizing api sequences with conditionals and loops. In *ICST*, 2018.

[159] Tingting Yu and Michael Pradel. Pinpointing and repairing performance bottlenecks in concurrent programs. *Empirical Software Engineering*, 2017.

[160] Pamela Zave. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.*, 2012.

[161] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *FSE*, 1999.

[162] Andreas Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.

[163] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *ICSE*, 2006.