The Dissertation Committee for Allison Sullivan
certifies that this is the approved version of the following dissertation:

Automated Testing and Sketching of Alloy Models

Committee:

Sarfraz Khurshid, Supervisor

Milos Gligoric

Christine Julien

Elizabeth Leonard

Dewayne Perry

Automated Testing and Sketching of Alloy Models

by

Allison Sullivan, B.S.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2017

To my husband, Bahji Ballard, my parents, Lori and Brian Sullivan, and my grandparents, Arlene Wilding (R.I.P.), Thomas Sullivan (R.I.P) and Eileen Sullivan, for their unconditional love and support.

Acknowledgments

I would like to thank the multitude of people who have helped me throughout my journey as a PhD student. First, I would like to sincerely thank my supervisor, Dr. Sarfraz Khurshid, for his support and guidance over the years. Sarfraz always goes above and beyond for any of his students. For me, as family health issues and even my own wedding popped up over the years of my PhD studies, Sarfraz never once hesitated to support and accommodate me as needed. I am thankful to the time he took to help me throughout, from troubleshooting ideas to 6:59 am paper submissions. I would also like to extend my appreciation to my committee members, Dr. Milos Gilgoric, Dr. Christine Julien, Dr. Elizabeth Leonard, and Dr. Dewayne E. Perry, for their invaluable guidance. In particular, I am grateful to Dr. Elizabeth Leonard for her helpful discussions and feedback, as well as her overall support from being a mentor for me during my internship at the Naval Research Laboratory to being one of my committee members.

I am thankful to my peers – current and former – in the Software Verification, Vaildiation and Testing group: Hayes Converse, Nima Dini, Shiyu Dong, Lisa Hua, Rui Qiu, Srinivasan Raghavendra, Kaiyuan Wang, Cagdas Yelen, Razieh Nokhbeh-Zaeem, Lingming Zhang, and Mengshi Zhang. Over the years, everyone in the group has always been welcoming and supportive, without hesitation. I'd like to extend a special thanks to Lisa and Hayes for their feedback as I prepared for my defense and for their help this past year. I'd also like to thank Melanie Gulick, the graduate coordinator for Electrical and Computer Engineering department, who is always prepared for anything and supportive through everything. I never once worried that I would be submitting the wrong paperwork or missing a deadline.

I'd like to extended my gratitude to my coauthors over the years: Razieh Nokhbeh-Zaeem, Kaiyuan Wang, and Dr. Darko Marinov. I am thankful to all the helpful discussions with Razieh as I was starting out as a graduate student and I am thankful to have had someone to help guide by example during a time period full of change. I am further thankful to fellow graduate student from the SVVAT group, Kaiyuan Wang, who has helped me flesh out research ideas (and sat through *all* my practice talks without complaint). I am thankful to Darko for his willingness to provide feedback over my research efforts and his invaluable perspective. Together with Kaiyuan, Darko, and Sarfraz, I will have fond memories of late night Skype calls about the trials and tribulations of sketching Alloy models while juggling 3 time-zones.

Most importantly, I'd like to thank my family and friends for all their love and support. To my close friends, Amber Mixon, Traci Overstreet and Samantha Pizzini, thank you for everything over the past few years from being my bridesmaids to being my support system. Your friendships have never failed to lighten up my days. In particular I'd like to thank my best friend, Traci Overstreet, who as an English teacher with no programming experience has dutifully proofread my papers over the years. She is one in a billion, someone I can draw strength from and I'm thankful for her guidance (read: she is *always* right), love and support over the past 10 years.

To my husband, Bahji Ballard, thank you for everything from providing a loving environment to taking on extra tasks when I had a deadline creeping up to sitting through all of my excited research rambles with a blank but loving look. To my parents, thank you for every opportunity you provided me with all my life and for every opportunity that you supported me through. To my grandparents, whom were and are endlessly proud of me, may you rest in peace. To my extended family, thank you for your unwavering support and eagerness to hear about my graduate school adventures. Lastly, a specical shout out to Maya Kaul and her car for literally making this submission possible.

This work was funded in part by the Cockrell Foundation through their bestowment of the Virginia and Ernest Cockrell, Jr. Fellowship and by the National Science Foundation under Grant Nos.: CCF-0845628, CNS-0958231, CCF-1319688, CNS-1218232, and CNS-1239498.

Automated Testing and Sketching of Alloy Models

Publication No. __________________

Allison Sullivan, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Sarfraz Khurshid

Models of software systems, e.g., designs, play an important role in the development of reliable and dependable systems. However, writing correct designs is hard. What makes formulating desired design properties particularly hard is the common lack of intuitive and effective techniques for validating their correctness. Despite significant advances in developing notations and languages for writing designs, techniques for validating them are often not as advanced and pose an undue burden on the users.

Our thesis is that some foundational and widely used techniques in software testing – the most common methodology for validating quality of *code* – can provide a basis to develop a familiar and effective new approach for checking the correctness of *designs*. We lay a new foundation for *testing* designs in the traditional spirit of testing code. Our specific focus is the Alloy language, which is particularly useful for building models of software systems.

Alloy is a first-order language based on relations and is supported by a SAT-based analysis tool, which provides a natural backend for building new analyses for Alloy. In recent work, we defined a foundation for testing Alloy models in the spirit of the widely practiced unit testing methodology popularized by the xUnit family of frameworks, such as JUnit for Java. Specifically, AUnit, our testing framework for Alloy, defines test cases, test outcomes (pass/fail), and model coverage, and forms a foundation that enables development of new testing techniques for Alloy.

To provide a more robust validation environment for Alloy, we build on the AUnit foundation in four primary ways. One, we introduce test generation algorithms, which automate creation of test inputs, which is traditionally one of the most costly steps in testing. Two, we introduce synthesis of parts of Alloy models using *sketching* by enumeration and constraint checking, where the user writes a partial Alloy model and outlines the expected behavior using tests, and our sketching framework completes the Alloy model for the user. Three, we investigate optimizations to improve the efficacy of our core model sketching techniques targeting improved scalability. Four, we introduce a second approach for sketching Alloy models that incorporates attributes from our test generation efforts as well as our initial sketching framework and uses equivalent formulas to outline expected behavior rather than tests. To evaluate our techniques, we use a suite of well-studied Alloy models, including some from Alloy's standard distribution, as well as models written by graduate students as part of their homework.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software designs play a key role in the development of reliable and dependable systems. In particular, modeling software designs enable developers to reason over the system as a whole before development and without focusing on the implementation level details, resulting in earlier detection of high level design faults. Recent years have seen much progress in the introduction of and support for declarative languages, which are well suited for conveying software designs [2, 3, 69]. However, declarative languages often come with steep learning curves for beginning users, in part because declarative languages are typically very different in nature from commonly used imperative languages such as Java and C++. Additionally, even for advanced users, writing correct models in hard, as these languages often lack robust integrated development environments and testing-oriented tool support. For instance, there is no equivalent of Eclipse or Net-Beans for Java in the declarative setting. As a direct consequence, model validation is ad-hoc, where users over time develop their own tips and tricks for checking their models, but there is no set testing infrastructure for end users to leverage. Together, these factors place a burden on the end-user and limit their ability to write correct models, eroding some of the benefits to these languages, such as the early detection of design faults.

Our thesis is that some foundational and widely used techniques in software testing – the most common methodology for validating quality of *code* – can provide a basis to develop a familiar and effective new approach for checking the correctness of *designs*. We lay a new foundation for *testing* designs in the traditional spirit of testing code. Our specific focus is the Alloy language, which is particularly useful for building models of software systems. Alloy is a first-order language based on relations and is supported by a SAT-based analysis tool, which provides a natural back-end for building new analyses for Alloy. In recent work [70, 71], we defined a foundation for testing Alloy models, AUnit, in the spirit of the widely practiced unit testing methodology popularized by the xUnit family of frameworks, such as JUnit for Java. Specifically, AUnit defines test cases, test outcomes (pass/fail), and model coverage, and forms a foundation that enables development of new testing techniques for Alloy.

To further provide a validation environment in Alloy, we build on our AUnit foundation in four primary ways. One, we introduce test generation algorithms, which automate creation of test inputs, which is traditionally one of the most costly steps in testing [71]. Two, we introduce synthesis of parts of Alloy models using *sketching* based on tests, where the user does not need to try and specify the entirety of the Alloy model. Instead, the user writes incomplete Alloy models and provides tests that represent "examples" (or counterexamples) of expected design properties which our sketching approach uses to produces complete models [75]. Three, we investigate optimizations for our sketching technique, focusing on improving scalability. Four, we introduce a second approach to sketching Alloy models that is formula-based rather than test-based and utilizes existing ad-hoc validation practices for Alloy. To evaluate our techniques, we use a suite of well-studied Alloy models, including some from Alloy's standard distribution, as well as models written by graduate students as part of their homework.

1.1 The Need for Testing in Alloy

The Alloy language offers expressive operators, e.g. quantification and transitive closure, that allow succinct formulation of complex properties. However, the expressiveness and succinctness can make Alloy models look deceptively simple. Formulating them correctly and reasoning their correctness can be quite challenging, especially for non-experts.

A key strength of Alloy is the Alloy Analyzer [31, 73], an automatic tool for *scope-bounded* reasoning where the analysis results hold for the given *scope*, i.e., bound on the universe of discourse. Alloy users write *commands*, which take two forms: (1) *simulation*, where the analyzer finds an *instance*, i.e., a valuation to the relations in the model such that the formula evaluates to true; and (2) *checking*, where the analyzer finds a *counterexample*, i.e., a valuation such that the negation of the formula evaluates to true. Technically, the underlying analysis for both forms is the same – solving *logical constraints.* The analyzer translates the Alloy model to a *propositional formula* with respect to the scope, uses off-the-shelf SAT solvers to solve it, and translates SAT solutions to Alloy instances or counterexamples.

Since models written in Alloy are logical constraints, Alloy models have two basic kinds of faults – *underconstraint*, where the formula allows valuations that the user wanted to rule out; and *overconstraint*, where the formula rules out valuations that the user wanted to allow. Alloy users employ two basic ad-hoc methods to validate their models. One, they use simulation to enumerate and inspect valuations to detect if some expected ones are missing or some invalid ones are present. Two, they use checking to validate expected properties between different formulas, e.g., checking (bounded) equivalence between two definitions that they expect are equivalent.

However, there are notable limitations to the existing ad-hoc validation techniques for Alloy. For simulation, the approach is limited by: (1) the order in which the Alloy Analyzer enumerates valuations is determined by the SAT solver, (2) the number of valuations to inspect can be hundreds, thousands or even hundreds of thousands and (3) despite the challenges of the order and size of valuations to inspect, the developer needs to notice either the absence of an instance or the incorrect structure of instance(s). For checking, the user needs to ensure that the check itself is not building false confidence in the correctness of the model. For example, when checking a logical relation, say implication between formulas f and g, i.e., $f \Rightarrow g$, it is standard practice to increase the scope and re-run the analyzer to increase confidence that the implication indeed holds; however, if f is overconstrained and simply false, increasing the scope only leads to a false increase in confidence. Similarly, when simulating

$f \wedge g$, and finding a solution, the user may inadvertently fail to notice that g is true but only vacuously.

Furthermore, validating the correctness of an Alloy model is conceptually very different from the widely used practice of *testing* imperative programs, which is conceptually simple: create some inputs (with respect to some coverage criterion or otherwise), run the program against them, and check the outputs. Before providing tool support for testing Alloy models, we first need to define conceptually what a test case and test execution should be within Alloy's declarative setting, in which we do not follow a sequential execution of program statements, but rather explore the space of all possible solutions allowed by the program. To address these challenges, our previous work [72], introduced basic definitions for unit tests, test execution, and model coverage for Alloy to lay the foundation of testing Alloy models in the traditional spirit of testing. In Chapter 2, we will outline these concepts in detail through an illustrative example in which we step through building, revealing and correcting a faulty Alloy model.

1.2 Dissertation Overview

1.2.1 AUnit Test Generation

We introduce two new approaches for automated test generation that create test suites following the spirit of traditional black-box and white-box testing. The black-box test generation technique, $AGen_{BB}$, creates suites that include all instances in the given scope and brings the spirit of *bounded exhaus-*

tive testing for imperative programs [8, 48] to Alloy models. The white-box test generation technique, $AGen_{WB}$, brings the spirit of *coverage-directed input generation* for imperative programs [9, 23, 59] to Alloy models. $AGen_{WB}$ reduces the problem of directed test generation for Alloy to constraint solving where model coverage requirements (introduced by AUnit) are part of the constraint. $AGen_{WB}$ iteratively builds a minimal set of (non-isomorphic) tests to meet the chosen criterion. Both techniques use Alloy's SAT-based backend for test generation and can be adapted to create suites based on different solving *strategies* [45, 52].

1.2.2 Enumeration-Based Sketching of Alloy Models Using Test Valuations

We introduce the first approach for *sketching* Alloy models, where the user does not need to attempt to write complete Alloy models. Our key insight is that valid and invalid valuations enable a user to specify the expected behavior of an Alloy model, providing a foundation for *sketching* Alloy models. To sketch an Alloy model, the user writes a *partial* Alloy model with *holes* and provides some valid and invalid valuations for the desired model, and the sketching infrastructure completes the partial model with respect to the given valuations. Our technique, $ASketch_{Eval}$, iteratively explores the space of all possible solutions to the sketch, using constraint *checking* – rather than constraint *solving* – and utilizes various optimizations to efficiently search this space and report a solution.

1.2.3 Exploring Refinements To Enumeration-Based Sketching of Alloy Models

We introduce two optimizations focused on improving $ASketch_{Eval}$, our enumeration-based sketching technique, which has shown to be an effective technique for sketching Alloy models. First, we outline a multi-threaded implementation of $ASketch_{Eval}$ and evaluate the effectiveness of a parallel approach to exploring the solution space. Second, we perform an exploratory study of the impact the order of test valuations – one of the key user inputs to $ASketch_{Eval}$ – has on the run-time. The outcomes of the study are used to derive insight into both the overall desirable attributes of a sketching test suite as well as a best practices approach for selectively ordering a test suite for sketching.

1.2.4 Formula-Based Sketching of Alloy Models

Prior to the introduction of AUnit, Alloy users leverage multiple adhoc methods for validating their models, typically related to leverage existing Alloy infrastructure such as instance enumeration or assertion checks for the existence of expected logical relationships between formulas. This latter approach often includes the use of *equivalence* checks between two formulas the user expects to be equivalent. For formula-based sketching, we utilize this existing validation practice and introduce $ASketch_{Equiv}$, which instead of requiring a user to provide a suite of test valuations, requires the end user to supply an equivalent formula. $ASketch_{Equiv}$ combines $AGen_{WB}$, test ordering observations, and $ASketch_{Eval}$ to provide a second approach to sketching Alloy models.

1.3 Contributions

We make the following contributions:

- **Automated test generation for Alloy** We introduce the idea of automated test generation for Alloy and techniques to generate tests for Alloy models in the spirit of traditional black-box and white-box test generation, titled $AGen_{BB}$ and $AGen_{WB}$ respectively.
- **Test Generation Experiments** We perform a two-fold evaluation to show the efficacy of our approach. One, we compute model coverage and mutation score (which is often considered as the strongest test adequacy criterion in imperative programs) for the generated tests. Two, we show our approach finds real faults in several subjects, including all 19 faulty models submitted as solutions to a homework question in a graduate course in our Department.
- ***ASketch*.** We introduce the idea of sketching Alloy models using valuations through the introduction of $ASketch_{Eval}$, an enumeration-based technique for sketching Alloy models based on evaluation alone (without any constraint solving) for completing sketches of Alloy models.
- **$ASketch_{Eval}$ Optimizations.** We introduce a suite of optimizations designed for relational expressions and logical formulas to improve the

efficacy of our techniques.

- ***ASketch* Experiments.** We present an experimental evaluation with small but intricate Alloy formulas to demonstrate that our enumeration based sketching technique introduces a promising approach for sketching Alloy models.

- **Multi-Threaded *ASketch*$_{Eval}$.** We present a multi-threaded approach to $ASketch_{Eval}$, which focuses on exploring the space of potential solutions to a sketch in parallel, and perform an evaluation over our core set of Alloy sketching models, focusing on the effectiveness of a parallel search over the solution space.

- **Random Test Order Experiments.** We investigate the effect of the test order, one of the main inputs to $ASketch_{Eval}$, on the overall run-time of the technique. Based on insights from 100 random test orders over all models in our *ASketch* evaluation model suite, we produce a series of guidelines for producing and ordering tests for sketching.

- **Formula-based sketching.** We present a second approach to sketching, $ASketch_{Equiv}$, in which sketching is driven by an equivalent formula rather than a collection of test valuations. We combine attributes of $AGen_{WB}$, $ASketch_{Eval}$ and test ordering observations to facilitate sketching.

- ***ASketch*$_{Equiv}$ Experiments.** We present an experimental evaluation over a small but robust suite of Alloy models, where the focus is on models with two distinct formulations of the same property. Our results explore the effectiveness of $AGen_{WB}$ generated test suites over one formula as a basis for sketching a formula with a different structure.

- **Embodiments.** For $AGen_{BB}$ and $AGen_{WB}$, we implement a prototype that embodies our techniques as well as the theoretical foundations introduced previously by AUnit, specifically to provide test case execution and code coverage computation in addition to automated test generation. We implement a prototype tool for *ASketch* that enables our range of optimizations and focuses primarily on the $ASketch_{Eval}$ portion of the *ASketch* framework. Attributes of both tools are combined together to form a prototype tool-set for $ASketch_{Equiv}$. Both sketching-based tool sets focus on finding the first solution to the sketch by default, but can additionally find some or all of the solutions. All tool-sets were used to facilitate our experiments across the various techniques.

1.4 Organization

The remainder of this dissertation is outlined as follows. In Chapter 2, we outline the background material for Alloy and AUnit through a guided example of building and correcting a singly-linked acyclic list model. Chapter 3 highlights the related work for both our test generation and sketching efforts for Alloy. Chapter 4 presents our two AUnit test generation techniques, as well as an evaluation comparing and contrasting the two approaches. Chapter 4 through Chapter 6 focus on sketching Alloy models. Chapter 5 introduces our

ASketch framework and details an enumeration-based approach to sketching that utilizes our AUnit test cases. Chapter 6 dives into two optimizations that serve to strictly improve the run-time of *ASketch* and provide insight into how to order and structure the input test suite. Chapter 7 presents a second approach to sketching Alloy models, which combines attributes from Chapters 4, 5 and 6 to develop a formula-based sketching approach. Finally, Chapter 8 outlines some proposed future research directions based on the work in this dissertation and Chapter 9 provides on overall summary of our work.

Chapter 2

Background

2.1 Alloy

An Alloy model consists of five kinds of *paragraphs*:

1. A **signature** (`sig`) declares a set of atoms, which can be viewed as user defined types. Each signature can declare 0 or more *relations*. In Alloy, relations are similar to fields of an object in the Object Oriented paradigm.

2. A **fact** (`fact`) is a formula that must always evaluate to true for any valid solution to the model. A fact paragraph allows a user to *explicitly* declare fact formulas. A fact can also be *implicitly* added to the model (e.g. multiplicity constraints from signature paragraphs).

3. A **predicate** (`pred`) is a named (and optionally parameterized) formula that can be *invoked*. A predicate can invoke other predicates, but the Alloy Analyzer does not support recursive predicates.

4. An **assertion** (`assert`) is a formula intended to be *checked* for validity.

5. A **command** (`run` or `check`) executes a predicate or checks an assertion. A `run` command directs the analyzer to find an *instance* to the

corresponding predicate, i.e., a solution to the *conjunction* of all fact formulas and the predicate formula. A `check` command directs the analyzer to find a *counterexample* to the corresponding assertion, i.e., a solution to the *conjunction* of all fact formulas and the *negation* of the assertion formula.

A command may invoke a formula *anonymously* by providing its body explicitly; the *empty* body "{}" represents the formula "`true`". Each command (implicitly or explicitly) specifies a scope – a bound on the universe of discourse – and the instances and counterexamples generated are within that scope. A command may include an expect clause: **`"expect 0"`** means no solution is expected and **`"expect 1"`** means some solution is expected.

The Alloy Analyzer *executes* a command for an Alloy model and reports the constraint-solving results. If instances or counterexamples are found, the user can inspect them (e.g., iterate through them) in a variety of different textual and graphical formats. The analyzer adds symmetry-breaking predicates to remove *isomorphic* solutions and reduce the total number of solutions [61]. While the analyzer's default symmetry breaking does not remove all isomorphisms, a number of previous approaches [20, 36] provide non-isomorphic enumeration for Alloy.

To see how these paragraphs piece together to form an Alloy model, we will step over one possible model for a singly linked list seen in Figure 2.1. The keyword `module` names the model, and allows for the model to be imported

```
module list.v.1.0
one sig List { header: lone Node }
sig Node { link: lone Node }
pred Acyclic { all n : Node | n !in n.^link }
```

Figure 2.1: A faulty Alloy model for a singly-linked list.

into other models. Two signature paragraphs (`one sig List` and `sig Node`) are declared, which introduces a singleton set of a list atom – due to the use of the multiplicity constraint `one` – and a set of node atoms respectively. Both signatures declare a binary relation. The relation `header` maps lists to nodes while `link` maps nodes to nodes. Both `header` and `link` restrict the mappings by using the keyword `lone`, e.g. `link` maps one `Node` atom to zero or one `Node` atom. The *predicate* `Acyclic` defines acyclicity to constrain that there is no cycle in the list. The body defines a universally quantified formula that reasons over the domain of all `Node` atoms (`all n : Node`) . Within the quantified formula, we use negation (`!`), set inclusion (`in`), relational composition (.) and transitive closure (^) to build a formula which states: "n is not in the set of nodes produced following 1 or more traversals using n's `link` relation", e.g. n is not reachable from itself.

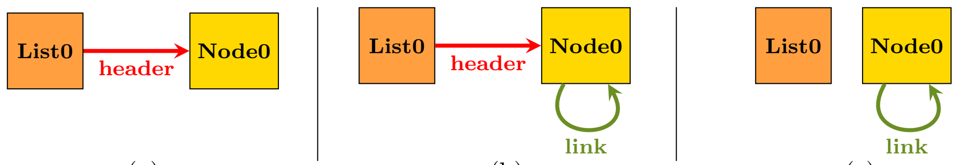
2.2 AUnit: A Testing Framework for Alloy

The section highlights AUnit, our testing framework for Alloy, and introduces the concepts of *test case*, *test execution*, and *test coverage*, which can be used to validate Alloy models by checking directly for both overconstraint and underconstraint faults, AUnit is inspired by the series of x-Unit frameworks frequently used to test imperative, object-oriented code. This subsections summarizes the efforts of two publications: "Towards a Test Automation Framework for Alloy" and "AUnit: A Testing Framework for Alloy."[1]

2.2.1 Test Case

A *test case* is a pair $\langle v, c\rangle$ where v is a valuation: a (potentially partial) assignment of values to sets and relations in the model; and c is either the *empty* command (i.e., $c = \epsilon$) or any valid Alloy command over the model, including but not limited to predicates and assertions in the model. Thus, a test may specify an assignment without stating any specific Alloy command. Moreover, a test may have commands other than those already given in the model. Additionally, the command acts as a label to indicate if the valuation v should be valid or invalid w.r.t. the existing model. Thus, Alloy users can create tests to directly check for underconstraint (when a given invalid valuation is allowed by the formula) and overconstraint (when a given valid valuation is not allowed by the formula).

[1] "Towards a Test Automation Framework for Alloy" in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software* by A. Sullivan, R. Nokhbeh-Zaeem, S. Khurshid, and D. Marinov. I was the lead student author of this paper and co-designed this work with S. Khurshid and R. Nokhbeh-Zaeem, with D. Marinov providing valuable guidance on writing the paper. This work is further explored (developed into a toolset and evaluated) in my master's thesis titled "AUnit: A Testing Framework for Alloy" in which I am the sole author.



(a) (b) (c)

```
pred Val1() {
   some disj List0: List {
      some disj Node0: Node {
         List = List0
         Node = Node0
         header = List0->Node0
         no link
   }
}
```

(d)

```
pred Val2() {
   some disj List0: List {
      some disj Node0: Node {
         List = List0
         Node = Node0
         header = List0->Node0
         link = Node0->Node0
   }
}
```

(e)

```
pred Val3() {
   some disj List0: List {
      some disj Node0: Node {
         List = List0
         Node = Node0
         no header
         link = Node0->Node0
   }
}
```

x

(f)

Figure 2.2: Graphical valuations: (a) (b) (c) and their corresponding textual representations(d) (e) (f).

2.2.2 Test Execution

A *test case* $t = \langle v, c \rangle$ *passes* if v is a solution to the constraint-solving problem for the command c. If $c = \epsilon$, a test passes if v is a solution to the constraint-solving problem for the command `"run {} for s"` where s is the scope required for v. Otherwise, t *fails*.

To highlight both the motivation for AUnit and how the concepts work in practice, we will step through producing and executing an AUnit test suite for our singly-linked list in Figure 2.1. Figure 2.2 depicts three valuation for our linked list model in both a graphical format and a textual format. The valuation simply states explicitly how to populate the sets and relations of the model. The keyword `disj` requires the elements in the associated set to be distinct, i.e. "`some disj Node0, Node1 :  Node` " means `Node0` and `Node1` have to be different `Node` atoms. Using the valuations in figure 2.2(a), we can make the following commands:

```
Test1: run { Val1 and Acyclic[] }
Test2: run { Val2 and !Acyclic[] }
Test3: run { Val3 and Acyclic[] }
```

Two of our tests, `Test1` and `Test3`, expect the valuation to satisfy `Acyclic`, while one test, `Test2`, expects the valuation to violate `Acyclic`. For `Test3`, although `Node0` has a self-loop, we expect `Val1` to be acyclic because `Node0` is not in the list. However, when we execute `Test3`, we discover the test has failed. Our model is *overconstrained*: we have ruled out a valuation we want to allow. When defining acylicity, we created a universally quantified formula whose domain is "`Node`", meaning our acyclic constraint (`n !in n.link`) has to be satisfied by all `Node` atoms, regardless of if they are in the list or not. To correct this, we can change our universally quantified formula's domain to reason over nodes in the list, producing:

```
module list.v.1.1
one sig List { header: lone Node }
sig Node { link: lone Node }
pred Acyclic { all n : List.header.*link | n !in n.^link }
```

Our new formula changes the domain of the quantified formula to be the set of nodes in the list by using reflexive transitive closure (`*`) to get the nodes reachable following **0** or more traversals along the header's link relation. Note, our use of reflexive transitive closure over transitive closure enables the list's header to be included in the domain, allowing for lists of size one to be acyclic.

2.2.3 Coverage

AUnit's coverage requirements require the tests to *exercise* various different values Alloy expressions and formulas may take as well as the different ways those values may arise. AUnit introduces coverage requirements for four Alloy elements: signatures, relations, expressions and formulas. Table 2.1 details these coverage requirements. The requirements outline different "shapes" for each construct (size and/or value). For signatures, relations and expressions, there are three requirements that each must evaluate to: an empty set, a singleton set, and a non-empty non-singleton set. On the other hand, formulas should evaluate to true and false. Furthermore, quantified formulas must be evaluated over an empty domain, a singleton domain, a non-empty non-singleton domain, and additionally require the formula bodies to also evaluate to true and false across these different settings.

AUnit formalizes these requirements into 8 coverage metrics. The metrics can be narrow such as predicate coverage, which focuses on the coverage of all expressions and formulas that appear in predicate paragraphs only, or wider ranging such as expression coverage, which looks at all expressions, regardless of their origin in the model. For the coverage reported in this dissertation,

Table 2.1: Coverage requirements defined by AUnit.

Alloy Construct	**Coverage Requirement**
Signature s	$\lvert s\rvert = 0$ $\lvert s\rvert = 1$ $\lvert s\rvert \geq 2$
Relation r	$\lvert r\rvert = 0$ $\lvert r\rvert = 1$ $\lvert r\rvert \geq 2$
Expression e	$\lvert e\rvert = 0$ $\lvert e\rvert = 1$ $\lvert e\rvert \geq 2$
Formula f	$f =$ true $f =$ false
Cardinality	$\lvert d\rvert = 0$ $\lvert d\rvert = 1$, $b =$ True $\lvert d\rvert = 1$, $b =$ False $\lvert d\rvert \geq 2$, $b =$ True $\lvert d\rvert \geq 2$, $b =$ True & False $\lvert d\rvert \geq 2$, $b =$ False

we will use *model coverage*: coverage calculated over all signatures, relations, expressions and formulas present anywhere in the model. Model coverage subsumes all other 7 coverage metrics.

Coverage for a test case t, where $t = \langle v, c\rangle \in T$, is computed as a set of maps $\langle \pi_t, \omega_t, \lambda_t\rangle$ where: (1) π_t maps each Alloy expression invoked by c to the set(s) of tuples it evaluates to for valuation v; (2) ω_t maps each Alloy formula invoked by c to the boolean value(s) it evaluates to for valuation v; and (3) λ_t maps each signature and relation to the set(s) of tuples it evaluates to for valuation v. Table 2.2 shows the model coverage for Test1.

Table 2.2: Coverage for Val1 over corrected singly-linked list model.

Alloy Construct	**Model `list.v.1.2` Constructs**	**Test1 Coverage**
Signature s	`List`	$\|s\| = 1$
	`Node`	$\|s\| = 1$
Relation r	`header`	$\|r\| = 1$
	`link`	$\|r\| = 0$
Expression e	`List.header.*link`	$\|e\| = 1$
	`n.~link`	$\|e\| = 0$
Formula f & Cardinality	`n !in n.~link`	$f =$ True
	`all n : List.header.*link | n !in n.~link`	$f =$ True
		$d = 1, b =$ True

Chapter 3

Related work

3.1 Automated Test Generation

This section highlights work done in similar spirits to our automated testing advancements for Alloy, outlined in Chapter 4. Efforts to introduce support for testing in its traditional form for Alloy can be found in a number of previous projects. For example, the Alloy tool-set has built in support for labeled commands, and allows running all commands; moreover, each command can use the `expect` clause to indicate whether a solution is expected or not. Indeed, we leverage this functionality to implement AUnit [72] tests. Moreover, Montaghami and Rayside's support for partial instances [50, 51] takes inspiration in part by the need to bring traditional testing ideas to Alloy. Furthermore, the Alloy analyzer's support for highlighting unsat cores helps with debugging faulty unsatisfiable formulas; we conjecture AUnit tests together with unsat cores provide an effective basis for introducing fault localization [34] and program repair [21] for Alloy. To our knowledge, AUnit is the only previous project that describes test cases, test execution, and model coverage (in their traditional spirit) for Alloy.

Symmetry-breaking [20, 37, 60] substantially reduces the number of solutions to inspect and plays a key role in traditional analyses using Alloy. Aluminum [52] limits the instances to *minimal* solutions to further ease the understanding of the models using instance inspection. Our test generation techniques are orthogonal to and can be applied in synergy with symmetry breaking, minimal instances, and other scenario exploration techniques [45].

Testing constraint programs in general has been addressed in previous work. For example, a test framework was built for the constraint language OPL which focuses on using an oracle model to derive tests that look for differences in behavior based on conformity properties and provides guidance for fault localization [42, 43]. Moreover, previous work introduced a reduction of testing UML models to satisfiability checking by encoding the model and a property of interest, and using SAT [64]. Other related efforts have focused on leveraging a range of UML diagrams (sequence, class, and activity diagrams) in conjunction with pre-/post-conditions and invariants in OCL to develop a few different methodologies for guiding test input generation, stimulating execution of UML models, and observing their behavior [14, 15, 56].

Numerous efforts have been made to introduce unit testing to the declarative language Prolog, and the collection of languages inspired by Prolog. SWI-Prolog is a commonly used Prolog implementation which includes a unit test framework called PLunit [77, 78] and provides tool support for test coverage and test reporting. For ECLiPSe Prolog, a constraint language based on Prolog, in addition to a bare bones unit testing framework, work has addressed automated test input generation [58]. Additionally, Mercury, a Prolog inspired language with a static type system, has support for unit tests in a declarative setting [12]. Moreover, in the context of functional programs, e.g., in Haskell, automated testing is common practice [10].

Our focus in Chapter 4 is on *testing* programs written in Alloy, a *declarative* language. There is a rich history of *using* declarative specifications to test *imperative* programs [24] with logical constraints playing an important role in systematic testing[8, 11, 22, 25, 27, 29, 39, 48, 53, 57].

While a primary role for tests is in bug finding, a number of recent projects have leveraged tests for automated debugging [21, 44, 46, 76] and program synthesis [19, 38, 65]. AUnit enables defining such approaches [75] for Alloy.

3.2 Program Sketching

In this section, we will highlight work done in similar fields to sketching Alloy models, the focus on Chapters 5, 6, and 7. In particular, we introduce the first approach to sketching Alloy. Program sketching [5, 28, 32, 62, 63, 65–68] is a form of program synthesis, which is a mature yet active research topic [7, 17–19, 26, 38, 41, 47, 54, 62]. 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, which is a well-studied field [13]. 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 takes as input a partial program in the Java-like Sketch language, and uses SAT and inductive synthesis in a counterexample-guided loop [65]. Sketch requires users to provide generators for candidate expressions for expression holes. The JSketch tool translates Java to Sketch to allow sketching Java programs [32]. Sketch4J [28] introduces an optimized backtracking search for completing Java sketches. Some tools focus on specific subclasses of programs to sketch, such as PSketch for concurrent data structures [67].

Using an enumeration-based approach to synthesize a program has been used before for Syntax-Guided Synthesis (SyGuS) [6]. EuSolver solves SyGuS domain problems by enumerating over all expressions enabled by the grammar until an expression satisfies the specification [6]. While our technique shares the spirit of optimizing through pruning redundant expressions, it works for a different programming paradigm and thus requires different means of generating expression and determining correctness.

Test-Driven Synthesis (TDS) iteratively builds a C# program such that it satisfies all tests [55]. TDS works over domain-specific languages that have a context-free grammar centered around the .NET framework. Component-based synthesis builds programs by having a component library used to form combinations of code within the language of the program being synthesized. More recent work on component-based synthesis has focused on using oracles related to I/O for programs that focus on loop-free work over inputs [33].

Our approach also 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 draw data structure manipulations than to write the code [63]. Our test valuations make use of a similar insight: an end user can more easily construct valid and invalid valuations than to write a complete model.

An approach for creating Alloy models using instances was introduced by aDeryaft [35] in the spirit of the Daikon [16] tool that uses a collection of known properties to check which of them hold with respect to given inputs. An approach for translating Alloy formulas to Java checks was introduced by MintEra [4]. Alchemy [40] defined a translation to database update operations and integrity constraints. AUnit [71, 72] recently defined the concepts of test case, test execution, and model coverage for unit testing of Alloy models in the spirit of popular xUnit frameworks for imperative languages. The test valuations that ASketch uses in the context of synthesis follow AUnit's definition of a test case.

Chapter 4

AUnit Test Generation

4.1 Overview

While AUnit introduces a testing foundation for Alloy, writing AUnit tests which reveal faults in a model can be tricky due to the nature of Alloy itself. As a declarative language, in Alloy, any behavior not ruled out by the model is valid. Indeed, this attribute is a key strength for using declarative languages to check software designs, as the language will explore all possible behavior, potentially revealing unintended restrictions (or lack thereof) of the design. However, this means AUnit test suites should be developed with the notion of exercising any and all assumptions made by the developers. For instance, in Chapter 2.2, our overconstrained fault was revealed by an AUnit test case in which a node was disconnected from a list. However, it is possible that a developer might assume Alloy will only generate nodes which are within the list. As a result, the developer would never generate `Test3`, thus never revealing the overconstrained fault.

Accordingly, to address the issue of test generation, this chapter introduces two new approaches for automated testing of Alloy models. Our approaches are based on creating test suites following the spirit of traditional black-box and white-box testing, respectively. The black-box test generation technique, $AGen_{BB}$, creates suites that include all (non-isomorphic) instances in the given scope and brings the spirit of *bounded exhaustive testing* for imperative programs [8, 48] to Alloy models. The white-box test generation technique, $AGen_{WB}$, brings the spirit of *coverage-directed input generation* for imperative programs [9, 23, 59] to Alloy models. $AGen_{WB}$ reduces the problem of directed test generation for Alloy to constraint solving where model coverage requirements (introduced by AUnit) are part of the constraint. Thus, any solution (if one exists) covers some previously uncovered requirement(s). $AGen_{WB}$ iteratively builds a minimal set of (non-isomorphic) tests to meet the chosen criterion. Both techniques use Alloy's SAT-based backend for test generation and can be adapted to create suites based on different solving *strategies* [45, 52].

This chapter is based on a paper titled "Automated Test Generation and Mutation Testing for Alloy" published in ICST 2017[1] and makes the following contributions:

- **Automated test generation for declarative models** – we introduce automated techniques to generate tests for Alloy models to support black-box and white-box test generation;

[1]A. Sullivan, K. Wang, R. Nokhbeh-Zaeem, and S. Khrushid "Automated Test Generation and Mutation Testing for Alloy" in *10th IEEE International Conference on Software Testing, Verification and Validation.* I was the lead student author for this paper in which I both designed, developed and evaluated the techniques for automated test generation (presented in this chapter) and designed the overall evaluation of the paper; fellow student Wang was the lead author for the mutation testing content (not presented in this chapter); with the remaining co-authors providing valuable guidance and feedback along the way.

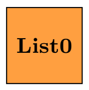
```
module list.v.2.1
one sig List { head: lone Node }
sig Node { link: lone Node }
pred Acyclic {
   some List.head => { some n: List.head.^link | no n.link }
}
```

Figure 4.1: A faulty Alloy model for a singly-linked list.

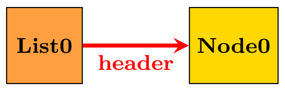
- **Implementation** – we implement a prototype that embodies our techniques as well as the theoretical foundations introduced previously by AUnit, specifically to provide test case execution and code coverage computation in addition to automated test generation; and
- **Experiments** – we perform a two-fold evaluation to show the efficacy of our approach. One, we compute model coverage and mutation score (which is often considered as the strongest test adequacy criterion in imperative programs) for the generated tests. Two, we show our approach finds real faults in several subjects, including all 19 faulty models submitted as solutions to a homework question in a graduate course in our Department.

4.2 Illustrative Example

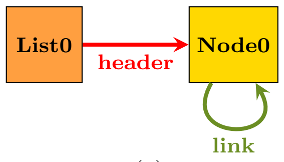
This section presents a small but representative example of a faulty Alloy model to refresh some key concepts in Alloy and AUnit, which will be used for our test generation techniques. Figure 4.1 (incorrectly) models an *acyclic*



(a)



(b)



(c)

```
pred Val1() {
   some disj List0: List {
      List = List0
      no Node
      no head
      no link
   }
}

T1: run { Val1 and Acyclic }
```

(d)

```
pred Val2() {
   some disj List0: List {
      some disj Node0: Node {
         List = List0
         Node = Node0
         head = List0 -> Node0
         no link
      }
   }
}

T2: run { Val2 and Acyclic }
```

(e)

```
pred Val3() {
   some disj List0: List {
      some disj Node0: Node {
         List = List0
         Node = Node0
         head = List0 -> Node0
         link = Node0 -> Node0
      }
   }
}

T2: run { Val3 and !Acyclic }
```

(f)

Figure 4.2: (a) Instance: empty list. (b) Non-instance: acyclic list with 1 node. (c) Non-instance: cyclic list with 1 node. (d) Passing test with valuation (a). (e) Failing test with valuation (b). (f) Passing test with valuation (c).

singly-linked list. Notice our acyclic constraint is different from section 2.1, which used universal quantification to expression acyclicity. This time, we use implication and existential quantification to express the same constraint. The *signature* (`sig`) declaration "`sig List`" introduces a set of list atoms; similarly "`sig Node`" introduces a set of node atoms. The keyword "`one`" declares `List` to be a singleton set. Each signature declaration also introduces a binary relation. The relation `header` maps lists to nodes. The relation `link` maps nodes to nodes. Both `header` and `link` are *partial* functions as declared by the keyword `lone`. The *predicate* (`pred`) `Acyclic` defines acyclicity. 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 keyword "`some`" represents existential quantification. The operator '`.`' is relational composition and '`^`' is transitive closure. The expression "`List.header.^link`" represents the set of nodes reachable from `List`'s header following 1+ traversals along `link`. The formula "`some E`" for expression `E` states that `E` is a non-empty set. Structuring `acyclic` as an implication additionally allows for the list to be empty, as without the implication, the existentially quantified formula requires there to be at least one `Node` atom in the `List`.

The *command* "`run Acyclic`" instructs the Alloy analyzer to create an *instance* for predicate `Acyclic` using the default *scope* – a bound on the universe of discourse – of 3. When the command is *executed*, the analyzer finds a valuation for `List`, `header`, `Node` and `link`, which satisfies the constraints in `Acyclic` and the *facts* in the model w.r.t. the given scope, i.e. considering up to 3 Node atoms, but 1 List atom (`one sig List`). Figure 4.2(a) graphically shows an example instance for "`run Acyclic`". Not all valuations are instances. Figures 4.2(b) and (c) are *not* instances for "`run Acyclic`" and thus will not be generated for this command. While (c) is expected to be a non-instance (since it has a cycle), (b) is a non-instance because of an overconstraint fault in our model.

To explore this fault, we can look at our AUnit tests. The predicates `Val1`, `Val2`, and `Val3` together with their respective labeled `run` commands `Test1`, `Test2`, and `Test3` represent three AUnit tests. Intuitively each predicate represents a test *input* and each command represents a test *oracle*. Note that `Val3` is not acyclic and `Test3` passes since the test oracle correctly identifies it (`!Acyclic`). `Test1` and `Test3` pass, but `Test2` fails and exposes a fault in the model. The valuation `Val2` is expected to be a valid acyclic list but the predicate `Acyclic` is overconstrained and erroneously disallows it; the fault is in the use of transitive closure (‘`^`’) in `List.header.^link`. This expression generates the set of Node reachable from 1 or more iterations over List’s `header`, but excludes the `header` itself. Therefore, since this expression appears inside an existentially quantified formula (`some`), this requires there to be at least one `Node` reachable from List’s `header` in order for the `Acyclic` constraint to be satisfied. To correct this, we can use *reflexive transitive closure* (‘`*`’) instead, which looks at *zero* or more iterations. Therefore, our corrected `Acyclic` predicate would be:

```
module list.v.2.2
one sig List { head: lone Node }
sig Node { link: lone Node }
pred Acyclic {
   some List.header => { some n: List.header.*link | no n.link }
}
```

For the Alloy model in Figure 4.1 and the default scope of 3, our black-box generation technique $AGen_{BB}$ creates 157 inputs while our coverage-driven test generation technique $AGen_{WB}$ creates 10 inputs. Both $AGen_{BB}$ and $AGen_{WB}$ produces all 3 valuations shown in Figure 4.2.

4.3 Automated Test Generation Using AUnit

This section presents our automated test generation approach. We introduce two basic techniques. $AGen_{BB}$ is a *black-box* technique which uses constraint solving to enumerate solutions for commands that execute (or check) existing predicates (or assertions) but does not directly utilize the way the Alloy model is written (Section 4.3.1). $AGen_{WB}$ is a *white-box* technique, which performs targeted test generation driven by AUnit's model coverage requirements (Section 4.3.2).

4.3.1 Enumeration-based Input Generation

Algorithm 1 embodies our enumeration directed approach; conceptually the algorithm systematically enumerates valuations by running the empty command (`run` {}), creates commands based on assertions and predicates (or their negations), and creates valuation-command pairs to form tests. The algorithm starts by executing the empty command ϵ over the given model. We selected the empty command because ϵ can be executed for any model. A naive approach would be to simply form test cases as follows: $\langle$enumerated instance i, $\epsilon\rangle$. For instance, we could build a test suite for our acyclic singly-linked list from Figure 4.1 as simply $\langle\langle$Figure 4.2(a), $\epsilon\rangle$, $\langle$Figure 4.2(b), $\epsilon\rangle$, $\langle$Figure 4.2(c), $\epsilon\rangle\rangle$. However, when a tester manually inspects the generated valuations to see if their shapes match expectations, the tester can only validate the behavior of valuations over the facts of the model. As a result, it is possible for the test suite to contain only valuations that are expected even though the model is actually faulty. To produce a more robust, informative test suite, we need to invoke other paragraphs in the model. Therefore, rather than using ϵ as the only command for our tests, we use ϵ to get a starting base of valuations.

Specifically, for each instance i enumerated by executing ϵ, we add a test case per each predicate and assertion paragraph present in the model, using the negation of the paragraph if the valuation fails the paragraph (the `i.eval(P)` call is false). Since all these valuations must satisfy the facts of the model, to reduce the number of overall tests, we do not explicitly generate tests that have ϵ as their command. Taking this into account, our new test suite for the model in Figure 4.1 is: ⟨⟨Figure 4.2(a), run Acyclic⟩, ⟨Figure 4.2(b), run Acyclic⟩, ⟨Figure 4.2(c), run !Acyclic⟩⟩.

While we now have a more robust test suite, all the tests have a common limitation: we never generate any tests in which the facts of the model are violated. Therefore, we create a second Alloy model, M', in which all the facts in the model are replaced with the disjunction of the negation of all the facts (`negateFacts`). Any instance satisfying M' violates at least one `fact` formula. However, the facts of the model limit the possible solution space significantly. If we repeat the process with M' (run ϵ, enumerate all instances, build up a test suite) we end up with extremely large state spaces, resulting in an unreasonable number of valuations to consider. Instead, we focus on enumerating one instance per each size in the scope. If ϵ is satisfiable for the tailored scope, a test case of the form ⟨first instance enumerated, `run` ϵ `expect 0`⟩ is added to the test suite. Recall, an `expect` clause is an Alloy construct used

Algorithm 1: $AGen_{BB}$ – Enumeration-based Input Generation.

```
input  : Alloy Model $M$, Scope $S$
output: Test Suite $T_{suite}$
$T_{suite} \leftarrow$ [];
Command empty $\leftarrow$ new Command(S, run {})
A4Solution solSpace $\leftarrow$ execute(M, empty)
if $solSpace.isSatisfiable$ then
    foreach $instance\ i\ in\ solSpace$ do
        foreach $Alloy\ paragraph\ P\ in\ M$ do
            if $P == PRED\ or\ P == ASSERT$ then
                if $i.eval(P)$ then
                    $T_{suite}$.add(new TestCase(i, $P$))
                else
                    $T_{suite}$.add(new TestCase(i, !$P$))
AlloyModel M' = negateFacts(M)
for $k \leftarrow 0$ to $S$ do
    Command empty $\leftarrow$ new Command(k, run {})
    solSpace $\leftarrow$ execute(M', empty)
    if $solSpace.isSatisfiable$ then
        $T_{suite}$.add(new TestCase(i, run {} expect 0))
return $T_{suite}$
```

to state whether a command should be satisfiable (`expect 1`) or not (`expect 0`). By appending `expect 0` to the empty command, the tester is informed that the associated valuations do not adhere to the facts of the model.

The technique can be implemented with any Alloy enumerator (Alloy Analyzer, Aluminum, TACO). We chose to implement Algorithm 1 to embody $AGen_{BB}$ using the Alloy Analyzer, which is the enumerator associated with the standard Alloy distribution. Although the SAT solver needs to be invoked when we first execute ϵ and enumerate instances, we can actually build up the remainder of the test suite without invoking the SAT solver: by using the Kodkod tool's `evaluator` API [73]. We invoke the `evaluator` by passing a test's valuation and an Alloy `formula` for a predicate or assertion. The `evaluator` returns `true` if the instance satisfies the `formula` and `false` otherwise. Therefore, we can use the `Evaluator` tp determine the shape of commands for our generated valuations, i.e. if the call to `Acyclic` should be negated or not. a

4.3.2 Coverage-Directed Input Generation

Our coverage-directed test generation technique $AGen_{WB}$ utilizes a feedback-loop that ensures each new test generated covers some previously unsatisfied coverage requirement. Specifically, the algorithm iteratively: solves the current formula to generate a test, obtains its coverage, and adds a logical constraint to update the current formula and capture some uncovered requirement(s). While these constraints place restrictions based on the coverage obtained for previously generated valuations, the *targeting* constraints do not include those valuations in any direct form. Moreover, the targeting constraints do not directly depend on the size of the previously generated valuations.

Alloy Analyzer's enumeration functionality does not guarantee that the next instance enumerated covers new criteria. Therefore, we introduce targeting constraints to be appended to the base model that will facilitate this ability. Table 4.1 shows the targeting constraint templates for Alloy constructs based on AUnit's coverage requirements. Signature, relation and expression coverage

Table 4.1: Targeting Constraint Templates.

Alloy Construct	**Coverage Requirement**	**Targeting Constraint**
Signature s	$\|s\| = 0$	$\{\#s = 0\}$
	$\|s\| = 1$	$\{\#s = 1\}$
	$\|s\| \geq 2$	$\{\#s > 1\}$
Relation r	$\|r\| = 0$	$\{\#r = 0\}$
	$\|r\| = 1$	$\{\#r = 1\}$
	$\|r\| \geq 2$	$\{\#r > 1\}$
Expression e	$\|e\| = 0$	$\{\#e = 0\}$
	$\|e\| = 1$	$\{\#e = 1\}$
	$\|e\| \geq 2$	$\{\#e > 1\}$
Formula f	f = true	{ f }
	f = false	{ !f }
Cardinality	$\|d\| = 0$	$\{\#d = 0\}$
	$\|d\| = 1$, b = True	$\{\#d = 1$ && $b\}$
	$\|d\| = 1$, b = False	$\{\#d = 1$ && $!b\}$
	$\|d\| \geq 2$, b = True	$\{\#d > 1$ && $b\}$
	$\|d\| \geq 2$, b = True & False	$\{\#d > 1$ && b some e: $d \mid b$ some e: $d \mid !b\}$
	$\|d\| \geq 2$, b = False	$\{\#d > 1$ && $!b\}$

requirements center around the size of the sets for the corresponding Alloy construct, while formula coverage requirements center around truth values. The cardinality subgroup outlines additional requirements for quantified formulas, where "d" is the domain and "b" is the formula in the body. Further details of the coverage requirements can be found elsewhere [70, 72]. To add a targeting constraint to a model, we write the constraint as a `fact`, guaranteeing its satisfaction. We call a model with a targeting constraint a *targeting model*.

There are different ways in which we can include coverage requirements in the model. One way is *unique-targeting*: iterate over all requirements and *target* each uncovered requirement by producing an associated targeting model, executing the empty command, ϵ, over the model, and using a satisfying instance, if any, as a valuation for a test case. An issue with unique-targeting is that each uncovered criterion is examined in isolation, and each infeasible criterion is only revealed through a unique invocation of SAT. To address this issue, we introduce *multi-targeting*, which explores multiple requirements together. Specifically, our targeting constraint is a disjunction of all unexplored requirements' targeting constraints. The multi-targeting template for a targeting constraint is:

```
For all unexplored requirements c1, ... cn:
   c1's targeting constraint or ... or cn's targeting constraint
```

For our running example model, the first multi-targeting constraint would cover all 35 requirements for the model: `(#List = 0) or (#List = 1) or (#List > 1) or ... or (no n.link) or !(no n.link)`.

Algorithm 2 encapsulates our coverage-driven approach. The algorithm runs as long as there is some coverage requirement which has not yet been explored, i.e. is not covered by a previous valuation or marked infeasible. First, all coverage requirements from the model are collected into a list (*criteria*), which will store the coverage landscape of the model as the algorithm runs. Next, a targeting model is created by appending the targeting constraint, which will either reflect the next criteria to target under the unique-targeting guideline or the disjunction of all remaining criteria to explore under the multi-targeting guideline (`getNextTC`). Then, ϵ is executed over the targeting model.

Algorithm 2: *$AGen_{WB}$* – Coverage-Directed Input Generation.

```
input  : Alloy Model $M$, Scope $S$
output: Test Suite $T_{suite}$
List criteria ← M.extractRequirements()
numExplored ← 0
AlloyModel targeting ← null
Command empty ← new Command(S, run {})
while numExplored != criteria.size() do
    TargetingConstraint target ← getNextTC()
    targeting ← appendTarget(M, target.formula)
    A4Solution solSpace ← execute(targeting, empty)
    if !solSpace.isSatisfiable() then
        targeting ← removeFacts(M)
        targeting ← appendTarget(targeting, target.formula)
        solSpace ← execute(targeting, empty)
    if solSpace.isSatisfiable() then
        Instance i ← solSpace.getInstance()
        foreach uncovered c in criteria do
            if i.eval(c.targetingConstraint()) then
                c.markCovered()
                numExplored++
                if i.eval(c.getPara()) then
                    $T_{suite}$.add(new TestCase(i, c.getPara()))
                else
                    $T_{suite}$.add(new TestCase(i, !c.getPara()))
    else
        foreach criteria c in target do
            c.markInfeasible()
            numExplored++
return $T_{suite}$
```

If the call is unsatisfiable, one of two possibilities hold: the requirements in the targeting constraint violate the facts of the model or the requirements are currently infeasible for the given scope. To determine which, we form a new targeting model with all facts removed except the targeting constraint (`removeFacts`) and execute ϵ. Should this call also be unsatisfiable, then all criteria being targeted cannot currently be covered w.r.t. the given scope and are marked infeasible.

However, should we produce a satisfiable solution space from either ϵ invocation, we have successfully targeted at least one new coverage requirement. We obtain the first instance, i, and analyze i for coverage information. We mark any requirement c that is covered for the first time as covered, removing c from the pool of requirements to target in future iterations. A requirement is known to be covered if the `evaluator` returns `true` for c's targeting constraint. Then, we generate a test case where the valuation is i and the command is either the predicate or assertion (or negation of when appropriate) that contains c. If c is located in a `fact` paragraph, the command is ϵ. For example, if Figure 4.2(a) is the first instance enumerated, the test ⟨Figure 4.2(a), run Acyclic⟩ would be created because the valuation in Figure 4.2(a) covers formulas and expressions in `Acyclic`, such as `#List.head = 0` and `some List.head = false`).

As with $AGen_{BB}$, $AGen_{WB}$ uses the `evaluator` to generate tests without additional invocations of SAT solvers. The `evaluator` is able to determine the shape of a test case's command when passed a requirement's origin paragraph (`c.getPara()`). Moreover, the `evaluator` is able to provide all coverage information. We can pass a test case's valuation and an Alloy `expression` or an Alloy `formula` depicting the coverage criterion. The call will return the out-

Table 4.2: Subject Alloy modules. For each subject, lines of code (*LOC*), number of predicates (# *p*), number of basic signatures (# *basic sig*), number of binary relations (# *bin rel*), number of ternary relations (# *ter rel*), number of variables, number of primary variables and number of clauses are shown. Pair (a, b) shows minimum (a) and maximum (b) over all commands in the model.

Module	**LOC**	**# p**	**# basic sig**	**# bin rel**	**# ter rel**	**vars**	**primary vars**	**clauses**
LinkedList	15	1	2	2	0	(195,351)	(24,30)	(274,521)
BinaryTree	20	1	1	2	0	(170,289)	(21)	(249,394)
FullTree	29	3	1	2	0	(170,429)	(21)	(249,1086)
Handshake	28	1	1	2	0	(420,488)	(34)	(704,926)
NQueens	26	2	1	2	0	(1509,2014)	(99,105)	(3858,5221)
Farmers	40	2	2	3	0	(791,888)	(64, 80)	(2147,2366)
Dijkstra	61	8	3	0	2	(296,1103)	(57)	(374,2066)

come of the expression or formula over the valuation thus indicating whether or not the criterion is covered.

4.4 Evaluation

This section presents an experimental evaluation of our test generation approaches. Our evaluation is two-fold. First, we use a suite of correct Alloy subjects to measure two common metrics used to evaluate test suites: (1) model coverage and (2) mutation score. Second, we use a suite of faulty subjects with *real* faults, including standard Alloy models and graduate homework submissions, to evaluate fault finding ability (Section 4.4.2) of test suites generated by our approaches.

4.4.1 Model coverage and mutation score

Subjects. We use seven correct Alloy models for evaluation. Four of these models, namely LinkedList, BinaryTree, FullTree and NQueens, are written by us, and the rest are from the standard Alloy distribution. These models range from simple illustrative examples like a singly-linked list to more complex examples like Dijkstra's mutual exclusion. LinkedList, BinaryTree, and FullTree capture the constraints of the corresponding data structures. NQueens solves the problem of how to place N number of queens on a chess board of size $N \times N$ without conflicts. Handshake encapsulates the Halmos handshake logic problem. Farmers outlines a common logic problem in which a person (the farmer) has to get three objects (fox, chicken, and grain) across a river without any of the objects eating another object. Lastly, Dijkstra captures Dijkstra's mutual exclusion algorithm to prevent deadlocks. Table 4.2 gives key characteristics of the subjects.

Model coverage results. Table 4.3 shows the details for various attributes related to generation and execution of the test suites, broken down by technique. **Model** is the Alloy model under test. **#Vals** is the number of valuations generated. **#SAT** is the number of calls made to the backend SAT solver during test generation. **#Tests** is the number of test cases generated. The SAT solver is invoked either when a command in the module is executed or the constraint to create the next instance is solved. Therefore, for both techniques, this value is the number of valuations plus the number of commands executed during test generation time. "$\mathbf{T}_{gen}$ **[ms]**" is the time to generate

Table 4.3: Test suite generation and execution.

AGen$_{BB}$							
Model	**#Vals**	**#SAT**	**#Tests**	**T$_{gen}$ [ms]**	**Mod Cov**	**T$_{exe}$ [ms]**	**T$_{cov}$[ms]**
LinkedList	1006	1012	4010	1	89.2%	870	9
BinaryTree	1004	1010	3004	1	91.5%	1824	2
FullTree	1004	1010	5004	1	90.5%	3410	4
Handshake	32	38	86	1	88.6%	186	15
NQueens	180	185	1888	3	84.0%	4585	56
Farmers	10	16	927	2	57.7%	205	21
Dijkstra	10	16	2893	2	64.7%	147	15

AGen$_{WB}$							
Model	**#Vals**	**#SAT**	**#Tests**	**T$_{gen}$ [ms]**	**Mod Cov**	**T$_{exe}$ [ms]**	**T$_{cov}$[ms]**
LinkedList	8	9	10	467	100.0%	2	2
BinaryTree	8	9	8	162	98.6%	1	2
FullTree	8	9	15	228	95.3%	3	3
Handshake	11	12	11	423	91.4%	5	5
NQueens	7	8	13	181	97.8%	2	3
Farmers	12	13	18	1171	87.37%	25	7
Dijkstra	12	13	14	2337	87.5%	15	7

the tests, starting with the first SAT invocation and ending once writing the test(s) to a file finishes. "**Mod Cov**" shows, as a percentage, the model coverage achieved by the tests. Model coverage considers all coverage requirements produce by all Alloy elements (signatures, relations, expressions and formulas) regardless of their location within the model. Since model coverage subsumes the other coverage metrics in AUnit [72], we only use model coverage for evaluation. "**T$_{exe}$ [ms]**" is the time to execute the tests, which is the time to check whether each valuation is an instance of its paired command. "**T$_{cov}$[ms]**" is the cumulative time that includes the test execution time and the time to calculate model coverage.

In all cases, *AGen$_{BB}$* produces the largest test suites, and takes the longest time to both generate and execute test suites. Since this technique can produce a very large number of tests, for our evaluation, we capped the number of instances that can be enumerated from any single command to 1000 for models without parameterized paragraphs, and 10 for models with parameterized paragraphs. As expected, $AGen_{BB}$ makes the most SAT calls. Furthermore, since Alloy's default symmetry-breaking does not remove all isomorphisms, some instances generate redundant tests. Naturally, since enumeration is oblivious of model coverage criteria, the resulting suites are not minimal for the level of coverage achieved. However, due to producing many valuations, $AGen_{BB}$ does, on average, have high model coverage. $AGen_{BB}$'s model coverage is hindered by exploring only a few valuations which violate at least one `fact` paragraph. Recall that enumerating all such valuations would likely create enormous test suites and this choice is a trade off to avoid a huge increase in test generation and execution times.

For all subjects, $AGen_{WB}$ produces a relatively small test suite: no more than 18 for any subject. Test generation, execution, and coverage computation altogether take less than a seconds for most subjects. In comparison to the other techniques, the coverage-directed generation technique produces tests that give the highest coverage for all subjects. Indeed, this technique is by design constructed to focus on increasing model coverage. While $AGen_{WB}$'s model coverage is not always reported as 100%, $AGen_{WB}$ does achieve the highest *possible* model coverage, as infeasible criteria are not currently factored out of the reported model coverage and the gaps in $AGen_{WB}$'s coverage is strictly due to infeasible criteria. A nice property of the technique is that it

Table 4.4: Mutation Testing.

Module	**# neq-mu**	**AGen$_{BB}$**		**AGen$_{WB}$**		**Mutation**	
		% k	**t[ms]**	**% k**	**t[ms]**	**% k**	**t[ms]**
LinkedList	27	92.6	120101	85.2	664	**100**	545
BinaryTree	59	59.3	137686	54.2	894	**100**	972
FullTree	82	70.7	429942	58.5	2426	**100**	1997
Handshake	78	76.9	341575	11.5	3363	**100**	3072
NQueens	93	87.1	2297711	69.9	4943	**100**	4099
Farmers	99	2.0	233811	2.0	2995	**100**	2986
Dijkstra	187	57.8	5039570	29.9	166154	**100**	94232

only generates non-isomorphic tests – by construction, there are no two tests that cover the same set of requirements. Note however, coverage-directed generation does not produce all non-isomorphic tests.

Mutation Testing Results Table 4.4 shows the results of mutation testing. For each subject, the table gives the number of non-equivalent mutants (# **neq-mu**), the mutation score (**% k**) and the time taken for running mutation testing algorithm (**t[ms]**) with respect to test suites generated using Alloy enumeration and coverage-directed generation. Additionally, the last column, shows the maximum reachable mutation score for each model, which is 100.

In all subjects, the Alloy enumeration technique uniformly gives higher mutation score compared to the coverage based technique. The exception is the `farmers` model, where both techniques only kill 2.0% of the non-equivalent mutants. These poor mutation scores are because most tests generated lead to unsatisfiable formulas in test execution because the Alloy enumeration technique generates predicate invocations with all permutations of parameters. Most parameter permutations make the tests trivially unsatisfiable for the

Table 4.5: Known Buggy Models – AUnit test generation fault finding ability.

Model	**AGen$_{BB}$**				**AGen$_{WB}$**			
	Fail	**#Vals**	**#SAT**	**#Tests**	**Fail**	**#Vals**	**#SAT**	**#Tests**
Farmers	**Y**	41	47	9769	**Y**	13	14	15
Dijkstra	**Y**	1004	1010	287004	**Y**	17	18	61

Table 4.6: Homework submissions – AUnit test generation fault finding ability.

AGen$_{BB}$				**AGen$_{WB}$**			
Fail	**#Vals**	**#SAT**	**#Tests**	**Fail**	**#Vals**	**#SAT**	**#Tests**
16/19	(223,504)	(227,509)	(26296,83753)	**19/19**	(13,18)	(14,19)	(16,47)

original model and all mutants, which means those tests do not kill any mutant. Another reason is that the `farmers` problem contains a very limited number of solutions given our scope of 5, which makes most tests unsatisfiable in the first place. The coverage-based technique kills only 11.5% of the mutants for the `handshake` model, mainly because the coverage based technique only needs to generate a small number of tests to achieve full coverage due to the nature of the model, which limits the ability to kill mutants.

4.4.2 Finding real faults

Known buggy models. We consider next two models that are faulty versions of two of our subjects in Section 4.4.1. Faulty `Farmers` and `Dijkstra` models are from Alloy Analyzer v4.1.10 that shipped these buggy versions; the current Alloy release includes the corrected versions after Alloy users discovered and reported the bugs. The faulty `Farmers` was *underconstrained*; the `CrossRiver` predicate checked the “eating” behavior on only one side of the river, enabling `Objects` which should have eaten each other to co-exist alone. In faulty `Dijkstra`, when the predicate `GrabbedInOrder` was invoked from `GrabOrRelease`, the model was *overconstrained* and only produced a few trivial instances. We generated test suites for each faulty model using both techniques ($AGen_{BB}$ and $AGen_{WB}$). If the test generation technique was able to produce a test which encapsulated the faulty behavior, then the bug is detected.

Table 4.5 summarizes the results of the two techniques. The column **Fail** shows if the technique creates a failing test and finds the fault. Both the $AGen_{BB}$ and the $AGen_{WB}$ multi-targeting find the faults in both models. $AGen_{BB}$ continues to create large test suites, which take significant time to produce and execute. For instance, the `Dijkstra` model contains no facts; therefore, the solution space for the empty command is substantially large: any instance that adheres to the signature declarations is valid. $AGen_{BB}$ is able to enumerate a diverse range of valuations, but each valuation produces multiple tests. The faulty `Farmers` model produces 41 valuations, but 9769 tests. $AGen_{WB}$ does not suffer from this limitation, but is still able to detect the faults.

Homework submissions. We consider next 20 homework submissions for one question that presented a partial model of a sorted, singly-linked-loop-list, where the last list node points to itself. The partial model contained 5 signature declarations, one fact skeleton and 5 predicate skeletons to be filled in by the students. Appendix B shows the partial model given in the homework. 19 of the 20 homework solutions were faulty. Using each technique, we produced a test suite over all 20 models. As with the known-buggy models, if the test suite produces a test which shows some buggy behavior, then the test suite has detected fault(s).

Table 4.6 summarizes the results of the two techniques. The column **Fail** shows the ratio of faulty models detected with respect to the total number of faulty models. Both techniques correctly identified the non-faulty student submission model. Details of the generated test suites are shown in pairs (a,b) showing the minimum and maximum values across test suites produced over all 19 submissions.

$AGen_{BB}$ detected bugs in a majority of the submissions, but was unable to detect a fault in 3 out of the 19 buggy models despite producing extremely large test suites. The loop-list has a number of predicates with parameters, which means a single valuation does not become a single test per command but a single test per unique combination of values for the command's parameter(s). As a result, for our evaluation, we "timed out" the technique after enumerating 500 instances, generally producing suites of size $> 80,000$. $AGen_{BB}$ struggled to detect faults located in `facts`, which, as expected, is a limitation of the $AGen_{BB}$ technique. The solution space for negated facts is typically much larger than the valid solution space. $AGen_{BB}$ had greater success at finding faults in the known-buggy models, whose faults were not in the facts of the model.

$AGen_{WB}$ was the most effective technique, finding a bug in *every* faulty student submission; moreover, it has small test suites, which can reasonably be inspected manually by end users to determine if the actual behavior matches the desired behavior. In the end, $AGen_{WB}$ is the only technique to detect a bug in every model in our suite of faulty subjects.

Our evaluation of AUnit test generation using homework submissions made us specifically appreciate AUnit. Aside from this evaluation, we had manually graded the submissions and some of the bugs found by AUnit were previously missed by us. These bugs were usually in relation to non-obvious *compounding* formulas, such as the impact a parameter has on the formulas within the paragraph. In Alloy, parameters are not allowed to be empty (by default), which can restrict the domains of the formulas present in the paragraph. Moreover, Alloy's expressive nature can make it difficult to reason about a group of formulas together, even if it is easy to understand them separately. Our $AGen_{WB}$ technique is able to automatically produce a test suite that has been shown to be capable of detecting bugs. While the end user will typically need to act as an oracle to the test suite (i.e. determining the `expect` clause), $AGen_{WB}$ produces representative but small test suites. Furthermore, since AUnit test cases are designed to be visualized, the end user can feasible step through an $AGen_{WB}$ test suite without much time or resources required.

4.5 Summary

We presented two novel approaches for automated testing of models written in Alloy – a well-known declarative, first-order language that is supported by a fully automatic SAT-based analysis engine. The two approaches introduces automated test generation that create test suites in the traditional spirit of black-box and separately white-box testing. The two approaches build on the basis defined previously by our AUnit framework, which introduced the idea of unit testing for Alloy in the spirit of unit testing for imperative languages. While test generation is a heavily studied problem with many solutions in the context of *imperative* languages, the key novelty of our work is to introduce and address these problems for the *declarative* programming paradigm, specifically for the Alloy language. Experimental results using several Alloy subjects, including those with real faults, demonstrate the efficacy of our framework.

We believe our work introduces a novel, yet conceptually familiar, effective way to validate the quality of Alloy models. Indeed, testing remains the most widely used methodology for validating quality of programs in imperative languages. We hope a well-founded testing methodology for models – written in declarative languages – can be equally useful for finding faults in models. We plan to develop our approach for other declarative languages and analysis tools in future work. We also believe testing in its traditional spirit has an important role to play in validation of declarative models.

Chapter 5

Enumeration-Based Sketching of Alloy Models Using Test Valuation

5.1 Overview

Declarative languages, such as Alloy, provide a means for specifying the design of a system, allowing for earlier validation of the system as a whole. Yet, declarative languages are often limited by their steep learning curves and lack of well-developed tool support. As a consequence, declarative languages often lack a formal testing framework, leaving users to adopt ad-hoc approaches to check for correctness. Together, these factors place an increasing burden on the end user, especially in the face of developing correct models. As a result, sketching, a form of program synthesis where users provide a *partial* program with holes and an outline of expected behavior of the program and a sketching tool outputs a completed program that meets the expectation(s), is one approach to improve the usability of declarative languages.

In this chapter, we introduce the first approach for *sketching* Alloy models, *ASketch*, where the user does not need to try and write complete models. Instead, the user writes a *partial* model with *holes* and provides some valid and invalid valuations that conceptually serve as test cases [72] for the desired model. These test valuations allow an end user to specify the expected behavior of an Alloy model, thus enabling sketching. *ASketch* focuses on sketching constructs of Alloy models, including relational expressions, logical operators, quantifiers, etc.

The main focus of this chapter is the introduction of $ASketch_{Eval}$, our enumeration-based technique to solve the sketch using constraint *checking* to simply evaluate—without any constraint solving—all possible candidate models one by one. Exhaustively considering all combinations of candidates for different holes to complete a sketch would require a prohibitively large number of calls to the `Evaluator`, although each call simply reduces to constant propagation; $ASketch_{Eval}$ optimizes sketching using a *per valuation* reduction on each set of candidate expressions and subformulas by evaluating (when possible) the candidates separately with respect to one test valuation (at a time) and removing duplicates, i.e., expressions or subformulas that evaluate to the same value for that valuation.

We perform an experimental evaluation of *ASketch* using 24 sketches derived from five core Alloy models. Experimental results show that *ASketch* can complete the models for sketches that can simultaneously have up to 3 expression holes and 3 operator holes. To highlight the complexity of the underlying problem, one formula, BinaryTree with 6 holes, has a state space of over 7.8 billion candidate Alloy models (3 operator holes with 4 candidates each and 3 expression holes with 496 semantically non-equivalent candidates each). $ASketch_{Eval}$, with all optimizations, makes 43,154 calls to the evaluator to explore 756 million candidate models before it finds a solution (w.r.t. 20 test valuations). Our results show that for all other cases except the binary tree model with 6 holes (which has the biggest search space), $ASketch_{Eval}$ finds a solution to the sketch quickly, with most sketches finishing in under 10 seconds.

This chapter is based on a paper titled "Sketching Alloy Models Using Test Valuations."[1] and makes the following contributions:

***ASketch*:** We introduce the idea of sketching Alloy models using test valuations. Our framework highlights the different problems for sketching an Alloy model: updating the Alloy input language, producing fragments, and solving the sketch.

$ASketch_{Eval}$: We introduce a technique based on evaluation alone (without constraint solving) for completing Alloy sketches.

Optimizations: We introduce a suite of optimizations designed for relational expressions and logical formulas to improve the efficacy of our techniques.

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

[1] "Sketching Alloy Models Using Test Valuations" is *submitted for peer-review* by A. Sullivan, K. Wang, D. Marinov, and S. Khurshid. This paper presents two sketching techniques: $ASketch_{Eval}$ (focus of this chapter) and $ASketch_{Solve}$ (not presented in this chapter). I was the lead student author for $ASketch_{Eval}$ – designing, developing, implementing and evaluating the technique with feedback from the coauthors. I worked closely with S. Khurshid in the design phase.

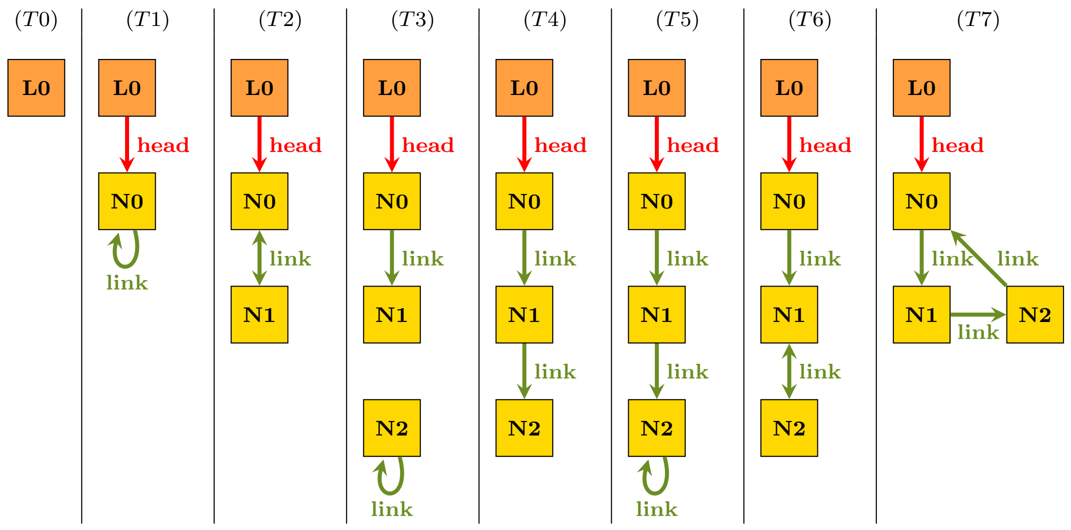


Figure 5.1: Eight valuations shown graphically. Of these, $T0$, $T3$, and $T4$ are valid valuations for the expected acyclicity property, and $T1$, $T2$, $T5$, $T6$, and $T7$ are invalid. $L0$ is the list atom. $N0$, $N1$, and $N2$ are node atoms.

ASketch brings the spirit of traditional program sketching [5, 28, 32, 62, 63, 65–68]—which is 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.

5.2 Example

To illustrate our *ASketch* approach, consider the following partial Alloy model for an acyclic singly linked list in Figure 5.2. The Alloy keyword `"module"` names the model, which can be included in another module. The *signature* (**`sig`**) declaration introduces a set of atoms and a user-defined type

Figure 5.2: A partial model of a singly-linked list.

```
module list v.3.0
one sig List { header: lone Node }
sig Node { link: lone Node }
pred Acyclic() { \Q\ n: Node | n \CO\ \E\ => n \CO\ \E\ }
```

in the Alloy model. A signature may optionally declare *fields*, i.e., relations. The signature `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 ≤ 1 `Node` atom. The signature `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 types of holes: one `\Q\` hole (quantifier hole), two `\CO\` holes (comparison operator hole), two `\E\` holes (expression hole). *ASketch* introduces these hole kinds to the Alloy grammar. 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 in the `Acyclic` predicate such that the formula we are sketching states that the nodes in the list form an acyclic structure. Figure 5.1 graphically illustrates eight test valuations for the model. Three of these valuations–$T0$, $T3$, and $T4$, are valid with respect to the expected acyclicity constraint, i.e., can be generated as instances by executing the command "`run Acyclic`" for a correct completion of the given predicate `Acyclic`. While five of these valuations – $T1$, $T2$, $T5$, $T6$, $T7$ – are invalid, i.e., cannot be generated by executing the command "`run Acyclic`". Note, valuation $T3$ is valid although its node $N2$ links to itself. $N2$ is not in the list, and the formula we are sketching should constrain only the nodes that are actually in the list to form an acyclic structure.

The user can provide the test valuations simply as Alloy predicates. For example, the following predicate represents test valuation $T0$ from Figure 5.1:

```
pred Test0() {
   some L0: List {
      List = L0
      no header
      no Node
      no link
      Acyclic[]
   }
}
```

This predicate uses an existentially quantified (`some`) formula to assign a value to the `List` set. Using the Alloy keyword `no`, the predicate declares the rest of the signatures and relations to be the empty set. The predicate invocation `Acyclic[]` labels the valuation as *valid* for the expected completion of the acyclicity constraint.

As another example, the following predicate represents test valuation $T2$ from Figure 5.1:

```
pred Test2() {
   some L0: List | some disj N0, N1: Node {
      List = L0
      header = L0->N0
      Node = N0 + N1
      link = N0->N1 + N1->N0
      !Acyclic[]
   }
}
```

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 completion of the acyclicity constraint.

Consider using *ASketch* to complete all five holes. Alloy allows 5 quantifiers for `\Q\` (`all`, `no`, `some`, `lone` and `one`) and 4 comparison operators for `\CO\` (`=`, `in`, `!=` and `!in`). Two are expression holes `\E\`, and using the optimized version of an expression generation technique outlined in [75], Alloy allows for 57 unique, non-equivalent expression candidates. Together, these substitutions produce a total of 259,920 candidate Alloy models. *ASketch*$_{Eval}$ searches threw this space to complete the sketch. Here is one solution *ASketch*$_{Eval}$ finds, with a universally quantified (`all`) formula:

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

The user may choose to enumerate more than one solution. For these 5 holes and 8 valuations depicted in Figure 5.1, $ASketch_{Eval}$ creates 4 formulas to complete the sketch:

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

All 4 formulas are equivalent and correctly represent our expected acyclicity constraint.

To complete the sketch, our optimized $ASketch_{Eval}$ takes only 0.6 seconds and invokes the Alloy evaluator 410 times. Moreover, $ASketch_{Eval}$ enumerates all 4 solutions on average in 0.4 seconds per solution and invokes the Alloy evaluator 128.5 times per solution.

5.3 *ASketch* Framework

We first present two aspects of the *ASketch* framework needed to allow sketching of Alloy models: updating the *ASketch* grammar to account for Alloy

```
expr ::= unOpE expr | expr binOp expr | expr arrowOp expr | "none" |
         "iden" | "univ" | "(" expr ")" | name | "\E\"

name ::= ("this" | ID) ["/" ID]*

quant ::= "all" | "no" | "some" | "lone" | "one" | "\Q\"

unOp ::= "no" | "some" | "lone" | "one" | "\UO\"

unOpF ::= "!" | "not" | "\UOF\"

logicOp ::= "||" | "or" | "&&" | "and" | "<=>" | "iff" | "=>" | "implies" |
            "\LO\"

compareOp ::= ["!"|"not"] actualCompareOp | "\CO\"

actualCompareOp ::= "=" | "in"

unOpE ::= "~" | "*" | "^" | "\UOE\"

binOp ::= "+" | "&" | "-" | "." | "->" | "\BO\"
```

Figure 5.3: Modified Alloy grammar used for holes in *ASketch*.

models with holes and the problem of how to generate Alloy fragments to use as candidates for the holes. Then, we describe how $ASketch_{Eval}$ determines which fragments complete the sketch to produce an Alloy model that satisfies all the given test valuations.

The input to *ASketch* is an Alloy model with holes. Our input language effectively extends the Alloy grammar with new syntactic constructs that represent holes. The current, full Alloy grammar is available online [49]; we follow an older exposition of Alloy [30] that also provided the semantics of the kernel Alloy language. Figure 5.3 shows the portions of the Alloy grammar that we have introduced holes for. For instance, we extend `quant` so the quantifier can

Table 5.1: Possible values for non-recursively defined holes.

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

be a hole `\Q\`and comparison operators (`compareOp`) include all operators from Alloy and also a hole `\CO\`. For all the kinds of holes depicted in Figure 5.3 minus `\E\`, we can use the concrete nature of the Alloy grammar to determine the substitution for each kind of hole. We will refer to these kinds of holes as *operator* holes. Table 5.1 shows the fragment substitutions for all operator kinds of holes.

Expression holes, `\E\`, present a unique challenge for generating fragments for substitutions for our sketching problem. Unlike operator holes, for expression holes, the Alloy grammar does not present a finite list of valid substitutions. However, the Alloy grammar does provide a blueprint for building expressions bottom up from the Alloy grammar, starting from smaller subexpressions and building larger ones by creating all the combinations of the subexpressions. While this avoids the need for an end user to provide the expression fragments, in practice, this leads to an exponentially large list of candidate expressions, which would make sketching infeasible. For instance, for our linked list model, building up expressions using just relational join could lead to an infinite generation such as: `n.link`, `n.link.link`, `n.link.link.link`, `n.link.link.link.link`, `...` , `n.link.` `...` `.link`. To even make generation of expressions feasible, bounds need to be set to determine a reasonable termination point.

In [75], we address all of these problems through the introduction of $ASketch_{Gen}$ which uses the following 4 bounds: (1) the max *depth* of the generated expressions, (2) the max *arity* of all generated expressions, (3) the *intended arity* of the expressions (if any), and (4) the max *number of operators* for each expression and two optimization approaches (static pruning and dynamic pruning) to generate robust but not prohibitively sized expression hole fragments. For our example model in Figure 5.2, $ASketch_{Gen}$ finds 1914 expressions, uses static pruning to reduce that to 67, and uses dynamic pruning to further reduce the static set to 57 candidate expressions. When we refer to $ASketch_{Gen}$ in the remainder of this chapter, we are referring to this technique.

5.3.1 $ASketch_{Eval}$: Evaluation-based sketching

$ASketch_{Eval}$ leverages the evaluator feature of the Alloy Analyzer, specifically the `Kodkod` tool [73] called `Evaluator`, to systematically check every possible *candidate* model (a completed model with all holes filled in) with respect to the provided test valuations. $ASketch_{Eval}$ takes as input an Alloy model with holes, a set of Alloy fragments for each hole generated by $ASketch_{Gen}$, and a set of user-provided test valuations (called simply *tests* in this context).

In the unoptimized version, $ASketch_{Eval}$ iterates over all possible combinations of fragments that can be formed to cover all holes, checks whether each combination satisfies all the provided tests, and if so, outputs the combination as a solution to the sketch. A key feature of $ASketch_{Eval}$ is that it requires no constraint solving. The main tool leveraged by the technique, the `Evaluator`, takes as input an Alloy expression/formula and an Alloy instance, and returns the value of the expression/formula with regards to the given instance. By using the `Evaluator`, which does not use the backend SAT solver, $ASketch_{Eval}$ only needs the valuation to check if each given test satisfies the constraints.

Specifically, $ASketch_{Eval}$ checks each candidate for each test by invoking the `Evaluator` created from the valuation either on the candidate or on its negation, depending on whether the test is marked valid or not. If the `Evaluator` returns `false`, the candidate does not properly complete the sketch; $ASketch_{Eval}$ immediately discards the candidate and moves to the next candidate. If a candidate satisfies all tests, $ASketch_{Eval}$ presents that candidate to the user as a solution. To illustrate, one incorrect candidate for the `Acyclic` predicate is:

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

The candidate satisfies the test in Figure 5.1($T0$); however, it does not satisfy the test in Figure 5.1($T1$), where it returns `false` rather than `true` for the valid test valuation. This candidate solution is immediately discarded, and $ASketch_{Eval}$ moves on to exploring the next candidate solution.

5.3.1.1 Optimizations

We improve on the unoptimized version with two optimizations that effectively re-use previous computations. The intuition is to not evaluate each and every expression in every candidate but to group expressions into equivalence classes (with respect to the provided tests) and to evaluate only one representative from each class. Multiple expressions often evaluate to the same *value*. For example, consider a test that depicts an empty list, i.e., no node reachable from `l.header`; the expressions `l.header`, `l.header.*link`, and `l.header.^link` all evaluate to the empty set. Therefore, $ASketch_{Eval}$ conceptually groups them together. In any given candidate where these three expressions could fill in the same hole, $ASketch_{Eval}$ evaluates only one expression over the empty list test valuation and applies that result across the other two expressions. We call this optimization $\text{ASketch}_{EvalExp}$.

Moreover, the equivalence classes need not be formed only at the level of individual expressions but can consider larger contexts of the parse subtrees that include expressions. Even if two expressions have different value for some test, e.g., E_1 ≡`l.header` and E_2 ≡`l.header.*link` have different values for non-empty lists, they may have the same value for the same test in a larger context, e.g., in `l in \E\`, both `l in` E_1 and `l in` E_2 evaluate to `true`. A good option in Alloy is to form equivalence classes over subformulas, because they evaluate to either `true` or `false` for each test. We call that optimization $\text{ASketch}_{EvalSub}$. An additional benefit of $ASketch_{EvalSub}$ is that it can create equivalence classes that combine multiple expression holes and even operator

Algorithm 3: Candidate Checker.

```
Input: ASketch model (M), tests (T), fragments for each hole (D)
Output: candidate solution
State: values: (Test,String) ↦ String // caches computed
  values ← {} // empty map
  cache ← {} // empty map (Test,String) ↦ Boolean
  foreach cand in buildCandidates(M, D) do
      foreach t in ts do
          rep ← ""
          if OPT = BASE then rep ← toString(cand)
          else if OPT = EXPR then
              foreach Expr hole h in M do
                  rep += memoExpression(t, h)
              foreach non-Expr hole h in M do
                  rep += cand.getValueForHole(h)
          else if OPT = SUBF then
              foreach subf in subformulas(M) do
                  rep += memoSubFormula(t, subf)
              foreach hole h not in a subformula do
                  rep += cand.getValueForHole(h)
          if !cache.containsKey(t, rep) then
              if !eval(t, cand) then
                  cache[(t, rep)] = false
                  break
              cache[(t, rep)] = true
          else if !cache[(t, rep)] then break
          if last test case then return cand
```

holes. To illustrate, for our list example, `\Q\ n : Node | n \CO\ \E\ => n \CO\ \E\`, $ASketch_{EvalExp}$ forms groups (for equivalence classes) for each `\E\` separately, whereas $ASketch_{EvalSub}$ first forms such groups but then also groups both `n \CO\ \E\` subformulas, before looking at the entire formula.

Algorithm 3 shows $ASketch_{Eval}$; the optimization level `OPT` is either `BASE`, `EXPR`, or `SUBF`. Implementation-wise, the algorithm then creates a pool of Kodkod `Evaluator` objects to avoid recreating them multiple times when the tests are checked in a nested loop. The algorithm first initializes `values`, a map that memoizes values of expressions and formulas (encoded as strings for easier presentation) for various tests. It also initializes a `cache` that memoizes encountered candidates (again, encoded as strings) to be looked up when an equivalent candidate model is encountered for a test more than once. The goal of these maps is to save calls to the Kodkod `Evaluator`, which can get expensive.

The algorithm builds all candidates by looping over all combinations of the provided fragments for each hole. For each combination, it produces a complete Alloy model, `cand`. For each candidate `cand`, the algorithm checks tests. For each test t, the algorithm first builds a representation `rep` of the candidate. With no optimization, the representation is the entire candidate (hence no re-use of any results across candidates). With an optimization, the representation includes the values previously computed and memoized in `values` for each expression over t (for $ASketch_{EvalExp}$) or for each subformula in the candidate (for $ASketch_{EvalSub}$). The `values` map is populated by the helper functions `memoExpression` and `memoSubFormula`, shown in Algorithm 4. An interesting aspect is handling free variables in expressions; `memoExpression` does that by trying all possible atoms for the domain of the expression, e.g., if the expression is `n.*link`, and `n`'s domain is `Node`, then `memoExpression` will compute the value of `n.*link` for all possible `Node` atoms in the scope.

The algorithm caches not only expression and formula values but the entire candidates, represented via the values of their expressions and formulas. If the representation `rep` is encountered for the first time, the algorithm calls the `Evaluator` and stores the result in `cache`. However, should `cache` contain `rep`, then the algorithm has already encountered a candidate that is equivalent (w.r.t. to the test), and the previous result from `cache` is reused. If the `Evaluator` returns `false` or the previous result was `false`, the candidate failed a test, and the loop exits without trying other tests. If the candidate does pass all tests, the candidate is printed as the solution.

In sum, our optimizations specifically target reducing calls to the `Evaluator`. Compared to SAT solving, those calls are cheap. However, they are not negligible; to evaluate a candidate, $ASketch_{Eval}$ needs to parse the candidate formulas – which can add to the cost when there are a large number of long, complex candidate expressions. By breaking the problem up into subproblems, which are much less expensive to solve for, we reduce the overall cost of the approach. Additionally, our optimizations are able to identify ways to prevent making – what are in the end – duplicate checks.

5.4 Experimental evaluation

We use 24 subjects derived from five core Alloy models: `LinkedList` from Section 5.2, `BinaryTree` models the acyclicity constraint of a binary tree, `Contains` checks whether a list contains an element, `Remove` models remov-

Algorithm 4: Helper Functions for Candidate Checker.

```
Input: test ($t$), Alloy expression ($e$) or formula ($f$)
Output: value of $e$ (or $f$) for $t$
    Function memoExpression(t, e)
        if !values.containsKey(t, toString(e)) then
            if !containsVariables(e) then
                values[(t, toString(e))] = eval(t, e)
            else
                foreach atom a of e's domain do
                    values[(t, toString(e))] += eval(t, e, a)
        return values[(t, toString(e))]
    Function memoSubFormula(t, f)
        k ← ""
        foreach Expr e in f.getExprs() do k += memoExpression(e)
        foreach Non-expression hole h do
            k += cand.getValueForHole(h)
        if !values.containsKey(t, k) then
            values[(t, k)] = eval(t, f)
        return values[(t, k)]
```

ing an element from a list, and `Dijkstra` models Dijkstra's mutual exclusion algorithm.

For each core model, we picked one predicate and created several sketching subjects by increasing the total number of holes (H#) 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 hole(s) affects our techniques. For example, for `LinkedList`, we identified 3 operator holes and 2 expression holes in the `Acyclic` predicate and produced these 5 variants:

```
\Q\ n: Node | n in List.header.*link => n !in n.^link
\Q\ n: Node | n \CO\ List.header.*link => n !in n.^link
```

```
\Q\ n: Node | n \CO\ \E\ => n !in n.^link
\Q\ n: Node | n \CO\ \E\ => n \CO\ n.^link
\Q\ n: Node | n \CO\ \E\ => n \CO\ \E\
```

We will refer to the different sketches produced over a base model as *model variants*. For each core model, we built up a test suite (*full*) for the model variant with the most holes such that the suite has at least 16 tests and all solutions found using *ASketch* are equivalent. Both the `BinaryTree` and `Remove` models have 20 tests, rather than 16. For the input fragments for operator holes, we use the Alloy grammar as outlined in Table 5.1. For expression holes, we use two optimized versions of $ASketch_{Gen}$ – static pruning and dynamic pruning – presented in [75]. Experimental results in [75] show a trade-off between the two approaches: static pruning finishes in seconds but produces larger fragments in comparison to dynamic which can take minutes but produces smaller fragments. For a robust look at $ASketch_{Eval}$, we will run experiments over both fragment generation techniques to see the impact of a sketch's fragment sizes on $ASketch_{Eval}$'s runtime. Table 5.2 shows both the size of the expression fragments produced for each expression hole and the time taken. The times reflected in the remainder of the evaluation section do not include $ASketch_{Gen}$'s run-time.

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

Table 5.2: $ASketch_{Gen}$ details

Model	Hole	Static	Dynamic
LinkedList	3,5	67	57
BinaryTree	2,4,6	617	496
Contains	1,3	621	227
Remove	1,3,4	1210	449
Dijkstra	2	9	9
	6	21	21

5.4.1 $ASketch_{Eval}$ results

$ASketch_{Eval}$ Performance. Table 5.3 shows the solving time for various sketching problems using different fragment generation approaches and test suite across all three $ASketch_{Eval}$ approaches. The columns A_{EB}, A_{EE} and A_{ES} show the time for the basic $ASketch_{Eval}$, $ASketch_{EvalExp}$, and $ASketch_{EvalSub}$, respectively. Our experiments use half-size and full-size test suites, where the half test suite is the first half of the full test suite. The half-size test suite can precisely sketch some of the model variants that have a few holes, but, with the exception of LinkedList, does not fully capture the sketching problem for the model variants with more holes. For all 24 sketches, using the full test suite allows only the desired model to be produced. Therefore, to validate our implementation, we used the Alloy Analyzer to check that every solution produced by our technique over the full test suite is *correct*, i.e. equivalent to our desired formula. Additionally, we checked that every solution generated in the experiments is *plausible* by checking that each solution passes all the tests associated with its generation.

As the number of holes increases, both $ASketch_{EvalExp}$ and $ASketch_{EvalSub}$

Table 5.3: *ASketch*$_{Eval}$ performance for finding the solution. Times are in seconds. ⊥ indicates a timeout (>15 min).

Model		Static Pruning						Dynamic Pruning					
		Half Test Suite			Full Test Suite			Half Test Suite			Full Test Suite		
		A$_{EB}$	A$_{EE}$	A$_{ES}$	A$_{EB}$	A$_{EE}$	A$_{ES}$	A$_{EB}$	A$_{EE}$	A$_{ES}$	A$_{EB}$	A$_{EE}$	A$_{ES}$
LinkedList	1H	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**
	2H	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**
	3H	0.4	0.2	**0.2**	0.4	0.3	**0.2**	0.4	0.2	**0.2**	0.4	0.2	**0.2**
	4H	1.3	0.5	**0.2**	1.5	0.5	**0.3**	1.3	0.4	**0.2**	1.5	0.5	**0.3**
	5H	16.2	3.6	**0.6**	15.9	4.0	**0.7**	14.1	3.4	**0.5**	13.7	3.9	**0.6**
BinaryTree	1H	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**
	2H	0.5	0.4	**0.3**	0.5	0.5	**0.4**	0.4	0.4	**0.3**	0.5	0.4	**0.3**
	3H	0.6	0.5	**0.3**	0.5	0.5	**0.4**	0.5	0.4	**0.3**	0.5	0.4	**0.3**
	4H	36.2	6.7	**1.2**	154.3	14.8	**2.7**	31.9	6.1	**0.9**	120.7	13.4	**2.0**
	5H	⊥	50.3	**12.2**	⊥	57.2	**12.4**	⊥	42.2	**7.8**	⊥	48.5	**9.3**
	6H	678.2	30.7	**9.0**	⊥	⊥	⊥	769.0	41.7	**7.4**	⊥	⊥	**776.4**
Contains	1H	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**
	2H	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**
	3H	**0.0**	0.0	0.1	0.3	0.3	**0.3**	**0.0**	0.0	0.1	0.2	0.2	**0.2**
Remove	1H	0.3	0.3	**0.3**	0.3	0.3	**0.3**	0.2	0.2	**0.2**	0.2	**0.1**	0.2
	2H	1.8	0.8	**0.8**	1.8	0.8	**0.7**	0.8	0.4	**0.4**	0.8	0.4	**0.3**
	3H	10.2	1.1	**1.1**	10.6	**1.1**	1.2	4.8	0.6	**0.6**	4.6	0.6	**0.6**
	4H	19.4	1.1	**0.9**	⊥	76.5	**57.3**	6.5	0.5	**0.4**	⊥	12.4	**9.5**
Dijkstra	1H	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**
	2H	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	**0.0**
	3H	0.1	0.1	**0.0**	0.1	0.1	**0.0**	0.1	0.1	**0.1**	0.1	0.1	**0.0**
	4H	0.1	0.1	**0.0**	0.1	0.1	**0.0**	0.1	0.1	**0.0**	0.1	0.1	**0.0**
	5H	0.2	0.2	**0.1**	0.2	0.1	**0.1**	0.2	0.2	**0.1**	0.2	0.1	**0.1**
	6H	5.5	4.0	**0.6**	6.5	4.3	**0.6**	6.4	3.9	**0.6**	6.6	4.7	**0.8**

make significant gains over the base algorithm; the base $ASketch_{Eval}$ outperforms the optimizations only for `Contains 3H`, where the optimizations provide a minor overhead given how quickly the solution is found. Additionally, table 5.3 shows that $ASketch_{EvalSub}$ outperforms $ASketch_{EvalExp}$. This behavior is expected, as a subformula would increase the chance for redundancy compared to an expression. All in all, across both static and dynamic pruning results, $ASketch_{EvalSub}$'s performance is relatively fast, with all but two sketches finishing in under a minute, and almost all finishing in under 10 seconds.

With an on-average fast run time and better performance than the other $ASketch_{Eval}$ algorithms, $ASketch_{EvalSub}$ is a solid initial approach to sketching Alloy models. There is one notable exception: `BinaryTree 6H`. For this sketch, $ASketch_{EvalSub}$, like the other versions, times out using static pruning, but is the only version of $ASketch_{Eval}$ to not time out using dynamic pruning. However, $ASketch_{EvalSub}$ takes 13 minutes to find a solution. This variant has the largest candidate solution space to explore, totaling over 7.8 billion for dynamic pruning (3 operator holes each at size 4 and 3 expression holes each at size 496) and requires the largest number of candidate solutions to be explored at over 756 million. However, large candidate solutions do not necessarily hurt $ASketch_{EvalSub}$'s performance: for static pruning, `Remove 4H` has over 5.3 billion candidates solutions and explores 311 million of them before finding a solution in 57.3 seconds.

Comparing Optimizations. Comparing the optimized versions of $ASketch_{Eval}$ to the base implementation, $ASketch_{EvalExp}$ and $ASketch_{EvalSub}$ are effective at

Table 5.4: Number of calls to the Evaluator using dynamic pruning expressions and the full test suite. ⊥ indicates a timeout (>15 min).

Model Variant	#Evaluator Calls							
	For Opt.		Checking Candidates			Total		
	A_{EE}	A_{ES}	A_{EB}	A_{EE}	A_{ES}	A_{EB}	A_{EE}	A_{ES}
LinkedList 5H	273	607	88473	10839	137	88473	11112	744
BinaryTree 6H	⊥	36397	⊥	⊥	6575	⊥	⊥	42972
Contains 3H	412	471	393	59	0	393	471	471
Remove 4H	2896	3864	⊥	11878	2450	⊥	14774	6314
Dijkstra 6H	195	537	137356	51830	1430	137356	52025	1967

decreasing the runtime, often significantly. For instance, consider `Remove 4H`, in which the base $ASketch_{Eval}$ technique times out while $ASketch_{EvalExp}$ finishes in 12.4 seconds (or 76.5 using static pruning) and $ASketch_{EvalSub}$ finishes in 9.5 seconds (or 57.3 seconds using static pruning). Digging deeper into the actual work done for the different algorithms, Table 5.4 looks at the number of calls made to the `Evaluator` to: (1) perform the optimizations – columns 2 and 3, (2) check if a candidate solution passes a test case – columns 4 - 6 and (3) the sum of the calls made for (1) and (2) – columns 7 - 9. Column 1 indicates which model variant we are considering: for each model, we selected the largest model variant.

`Contains 3H` represents an interesting corner case in which $ASketch_{EvalExp}$ and $ASketch_{EvalSub}$ make the same number of calls to the `Evaluator`. For this model variant, the entire sketch is one formula with no subformulas, $ASketch_{EvalExp}$ and $ASketch_{EvalSub}$ will end up doing the same amount of work. In particular, $ASketch_{EvalSub}$ does not need to use the `Evaluator` to check any candidates, as the representation produced by the optimization for a "subformula" captures the behavior of the candidate over the test valuation.

The number of `Evaluator` calls that $ASketch_{Eval}$ makes reduces as more advanced optimizations are used. Specifically, the basic algorithm makes the highest number of `Evaluator` calls and $ASketch_{EvalSub}$ has the smallest number of calls, with $ASketch_{EvalExp}$ in between. While $ASketch_{EvalExp}$ makes less calls to the `Evaluator` to perform the optimization, $ASketch_{EvalExp}$ makes more *overall* calls to the `Evaluator` than $ASketch_{EvalSub}$, including more of the expensive calls to check the candidate's behavior over a test valuation. In other words, $ASketch_{EvalExp}$ forms less equivalence classes over the candidate solutions, requiring more overall work to find a solution to the sketch. On the other hand, $ASketch_{EvalSub}$ makes more `Evaluator` calls to form *less* equivalence classes, resulting in less work and thus better performance. Overall, while the optimizations do extra work to determine a candidate's equivalence class, the trade-off of grouping candidates together to reduce the number of candidates which need to be *individually* checked across a test valuation is worth it, as seen by the difference in performance across the techniques in Table 5.3, especially as the complexity of the sketching problem grows.

In the end, as $ASketch_{EvalSub}$ is almost always the fastest, it is currently the best technique for sketching Alloy programs. The main strength of $ASketch_{EvalSub}$ is the optimization's ability to avoid checking redundant candidates, and instead directly applying the previous evaluation result. $ASketch_{EvalSub}$ finds more redundancy than $ASketch_{EvalExp}$, giving $ASketch_{EvalSub}$ consistently better performance across our range of models, test suites, and fragment generation approaches. Regardless of optimizations, $ASketch_{Eval}$'s run-time is dependent on finding a solution by exploring combinations. The more combinations there are, the longer the search may take. Future work can determine if different orderings of the generated expressions, different exploration of the holes, or different ordering of the tests can improve $ASketch_{Eval}$'s performance.

5.5 Summary

We introduced *ASketch*, the first approach for sketching Alloy models. Given a model with holes and some valid and invalid valuations, *ASketch* completes the model with respect to the valuations. The *ASketch* framework has three key components: extending the Alloy grammar, outlining how to generate a pool of candidate fragments for each hole, and exploring the resulting space of candidate models to find one that conforms to the valuations. *ASketch* introduces a suite of optimizations to efficiently explore very large spaces of candidate models. An experimental evaluation using a suite of subject sketches shows that *ASketch* introduces a promising approach for sketching Alloy models.

Chapter 6

Exploring Refinements To Enumeration-Based Sketching of Alloy Models

6.1 Overview

In Chapter 5, we introduced a sketching framework for Alloy, *ASketch*, in which the user does not need to focus on developing complete and correct Alloy models, but instead can provide partial Alloy models that *ASketch* completes for the end user. *ASketch* is based on the insight that our AUnit tests enable a user to outline the expected behavior of a model. To solve the sketch, we introduced $ASketch_{Eval}$, an enumeration-based approach that uses constraint *checking* to explore candidate models. The optimized version, $ASketch_{EvalSub}$, has proven to be a promising approach to sketching Alloy models (section 5.4, [75]).

A key attribute to make any sketching technique feasible for use is *scalability*. $ASketch_{EvalSub}$ is able to find solutions to almost all model variants in under 10 seconds. However, for the model variant with the largest solution space, `BinaryTree 6H`, $ASketch_{EvalSub}$ takes about 13 minutes to finish, which is significantly longer than the next highest runtime at 9.5 seconds for `Remove 4H`. Therefore, this chapter focuses on exploring avenues to improve the overall run-time of $ASketch_{Eval}$, motivated by improving `BinaryTree 6H`'s runtime.

For `BinaryTree 6H`, given our dynamic pruning list of fragments and the current order in which we explore the space of candidate models, $ASketch_{Eval}$ checks 756 million candidates before finding a solution. This highlights the main limitation of any enumeration-based technique: the technique may end up searching over a large number of candidates. If the order we explore candidates remains the same, while we can optimize the work done for a candidate, we cannot avoid the need to explore 756 million candidates to sketch `BinaryTree 6H`. To address this, our first refinement explores a multi-threaded implementation of $ASketch_{Eval}$, where we outline how to explore the solution space in parallel.

Notably, exploring hundreds of millions of candidates does *not* guarantee a long run time. In comparison to `BinaryTree 6H`, consider `Remove 4H` using static pruning fragments. $ASketch_{EvalSub}$ explores 311 million candidates – 2.4 times less candidates than `BinaryTree 6H` – but finds a solution in 57.4 seconds – 13.5 times faster than `BinaryTree 6H`. Why is `Remove` able to explore candidates faster than `BinaryTree`? Upon comparison, `Remove 4H` has *fewer* equivalence classes than `BinaryTree 6H` thus enabling the $ASketch_{Eval}$ to explore candidate models *faster*, as a whole. In section 5.4, we saw the benefit of fewer equivalence classes in the difference in performance between $ASketch_{EvalExp}$ and $ASketch_{EvalSub}$, where our *per candidate* level optimizations compound to given $ASketch_{EvalSub}$ a markedly better performance.

A series of improvements made at the implementation level of *AS*-

$ketch_{Eval}$ since its initial embodiment further supports this insight. Over time, our embodiment of $ASketch_{Eval}$ has improved, as we have found various redundancies in the implementation that could be removed. For instance, one such improvement is utilizing the fact that when we explore the next test candidate, not all of the subformulas change. As a result, we can keep a local reference to the representation of each subformula in the sketch, and only look for a new representation when some subformula differs from the previous candidate explored. This reduced the work per candidate solution, and served to reduce the run time over the `BinaryTree 6H` using $ASketch_{EvalSub}$ from 1078 seconds to 776.4 seconds. Accordingly, our second improvement to $ASketch_{Eval}$ focuses on improving the order of the test valuations, one of the main inputs to *ASketch*, which impacts the amount of work each candidate solution must do: the earlier a candidate is eliminated by a test, the less overall work the candidate does.

In total, this chapter makes the following contributions:

$ASketch_{Eval}$ Multi-Threaded: We introduce the idea of breaking the space of candidate models to be explored by $ASketch_{Eval}$ into a series of smaller domains to explore in parallel.

Exploring Test Order: We outline and provide the results of a random test order experiment, designed to determine the impact of the test order on the overall run-time of $ASketch_{Eval}$.

Guidelines for Test Valuations: Based on the random test order study, we present a series of guidelines for both the generation of a test suite as well as ordering a test suite for sketching purposes.

Experiments: We present an experimental evaluation of our multi-threaded technique and produce the random test orders runs over a small but intricate collection of Alloy formulas, previously used to evaluate $ASketch_{Eval}$.

The remainder of this chapter is organized as follows. Next, we will outline our subject Alloy models which will be used to evaluate both improvements to $ASketch_{Eval}$. Then, we will both step over and evaluate our multi-threaded implementation. Lastly, we will present our random test order experiment and provide an analysis of the results, culminating in a set guideline for generating and ordering test suites for sketching.

6.2 Subjects

As we explore different optimizations to $ASketch_{Eval}$, we will reason over results from five core models. Two models represent common data structures: `LinkedList` and `BinaryTree` capture the acyclic property of a singly-linked list and a well-formed binary tree respectively. Two models reason over a list in which node's contain an element value: `Contains` checks if a given element is in the list and `Remove` checks if a given element has been removed from the list. Lastly, `Dijkstra` captures the deadlock property that is prevented by Dijkstra's mutex ordering algorithm. All experiments were run for a scope of three. Sketches were formed by abstracting away one predicate from each model, iteratively building sketches by removing one valid construct each iteration, as we did in our experiments in section 5.4. For our experiments, we used dynamic pruning output from our $ASketch_{Gen}$ approach outlined in [75]. For the input test suites, we used our full test suite from our prior experiments, which encapsulates our sketching problem. For `LinkedList`, `Contains`, and `Dijkstra` the test suite consists of 16 tests, while `BinaryTree` and `Remove` have 20 tests. All experiments use the same set of subjects and test suites. The experiments were run on an Intel Core i7-4790K Quad-Core 4.0 GHz on Windows 7.

6.3 Multi-Threaded $\boldsymbol{ASketch}_{Eval}$

$ASketch_{Eval}$ is overall an enumeration technique which iterates over all possible combinations, until a solution is found. As a result, $ASketch_{Eval}$ lends itself to a multi-threaded algorithm, since the domain – the entire collection of unique candidate solutions – can be divided across threads and checked in parallel. In this section, we will outline a multi-threaded implementation evaluated using $ASketch_{EvalSub}$.

6.3.1 Approach

To demonstrate how a multi-threaded algorithm would work in the context of $ASketch_{Eval}$, we will first step over how $ASketch_{Eval}$ currently enumerates over the domain of all possible candidate models. Before sketching, for each hole, we determine the *rate* at which the hole should change the value it contributes from its fragment list, following the `findRateOfChange` function in algorithm 5. The first hole increment is set to 1, meaning that the candidate presented by "hole 0" will change every time. From there, a hole's "rate of change" is set to the multiple of all previous holes' sizes. The results are stored in an array, `rateOfChange`, which will be used by the main $ASketch_{Eval}$ algorithm to facilitate iterating over candidate models.

In detail, $ASketch_{Eval}$'s iteration is driven by a combination algorithm. To start, an outer loop iterates from zero to the total number of combinations, or until a solution is found and the loop is exited. For each iteration, an inner loop determines if any hole(s) should be incremented to the next candidate value by checking if the current counter, `k`, is evenly divisible by the increment at which a given hole should be changed, `rateOfChange[i]`. If so, then that hole is incremented to the next value, as outlined in the last loop of algorithm 6.

For a multi-threaded implementation, we will divide one range, [0, totalCombinations], into a series of smaller ranges to be sent to individual threads. Easily, these ranges can be set up to have no overlap; however, the challenge is in determining what candidate solution the start of the range depicts. Additionally, we want a simple approach that will not add significantly to the start up cost of the thread. Our methodology for getting the starting value is depicted in `findStartingValues` in algorithm 5. First, we determine the number of times the hole has changed the value it contributes, which can be determined by dividing the starting value by the increment at which the hole is meant to change. This number, `numChanges`, in theory shows how far into its iteration a given hole is. However, it is possible that a hole has looped over

Algorithm 5: Helper Algorithms for Multi-threading

```
State: holeSize = size of candidates for each hole
numHoles = number of holes
Output: rate of change for each hole stored in rateOfChange
Function findRateOfChange()
    long prevTotal = 1
    // find increment at which the hole should switch to a new
     candidate value
    for i = 0, i < numHoles, i++ do
        rateOfChange[i] = prevTotal;
        prevTotal *= holeSize[i];

Input: starting position of range (start)
Output: population of pos based on start
Function findStartingValues(start)
    // find the starting candidate the hole contributes
    for i = 0, i < numHoles, i++ do
        long numChanges = start / rateOfChange[i];
        pos[i] = numChanges % holeSize[i];
```

its size and is on a second(+) iteration; therefore, the starting contribution for a hole is calculated as the number of changes (`numChanges`) modulus the size of the fragment list for a hole (`holeSize[x]`). Algorithm 6 outlines the basic structure each thread follows to complete the sketch. First, `findStartValues` is invoked, then all the candidate solutions within the range sent to the thread are executed. The bottom portion of the algorithm outlines how the candidate solution is incremented to the next candidate within the range.

Algorithm 6: Multi-threading Outline

```
State: totalCombinations = total number of unique combinations
  to solve the sketch
pos ↦ lookup for the candidate contributed by each hole
rateOfChange ↦ lookup for the interval at which each hole
  changes its contribution
findStartingValues(start)
k ← start
while k < totalCombinations do
    cand ← buildCand(pos)
    // $ASketch_{Eval}$ sketching code here...
    k++
    for i = 0, i < numHoles, i++ do
        if k % rateOfChange[i] == 0 then
            pos[i] = pos[i] + 1;
            if pos[i] == holeSize[i] then pos[i] = 0;
```

6.3.2 Evaluation

Table 6.1 shows the results for running $ASketch_{EvalSub}$ over our 24 different model variants for 1 thread, 2 threads, 4 threads, and 8 threads respectively. The first two columns depict which model variant is being evaluated, while columns 3 through 6 show the runtime for the different number of threads. The runtime does not include the time to generate fragment lists, the work of $ASketch_{Gen}$, but does include the time to start up and terminate all threads.

Overall, our results show that a multi-threaded approach is a promising avenue for improving $ASketch_{Eval}$ runtime. Of note, we do not expect a direct decrease in time per thread: two threads will not necessarily speed up the

Table 6.1: Multi-threaded Experiment Results Using $ASketch_{EvalSub}$.

Model		**Number of Threads**			
		1	**2**	**4**	**8**
LinkedList	1H	0.01	0.03	0.06	0.06
	2H	0.01	0.03	0.06	0.07
	3H	0.05	0.06	0.06	0.08
	4H	0.07	0.07	0.07	0.08
	5H	0.14	0.10	0.10	0.11
BinaryTree	1H	0.02	0.04	0.09	0.12
	2H	0.14	0.14	0.12	0.11
	3H	0.13	0.10	0.09	0.08
	4H	0.90	0.53	0.42	0.23
	5H	6.49	3.77	2.64	1.67
	6H	728.66	418.25	267.23	172.77
Contains	1H	0.03	0.05	0.11	0.11
	2H	0.03	0.05	0.11	0.11
	3H	0.08	0.12	0.11	0.10
Remove	1H	0.09	0.10	0.14	0.13
	2H	0.18	0.18	0.17	0.14
	3H	0.27	0.22	0.20	0.19
	4H	7.53	4.18	2.74	1.45
Dijkstra	1H	0.04	0.07	0.15	0.18
	2H	0.04	0.07	0.15	0.17
	3H	0.04	0.07	0.15	0.17
	4H	0.04	0.07	0.16	0.18
	5H	0.05	0.07	0.16	0.18
	6H	0.29	0.21	0.26	0.31

runtime by a factor of two. The work of $ASketch_{Eval}$ is to iterate over a domain until a solution is found. Therefore, at a certain point through a thread, a solution will be found and all results will terminate. For our models which finish in less than a second, the multi-threaded approach does add a minor overhead, which can cause the multi-threaded approach to be slower, albeit by 100 to 200 milliseconds. This overhead is not large enough to deter the use of multi-threading. Furthermore, we can see the real benefit to a multi-threaded implementation when we look at any of the model variants which take longer than a second to solve: `BinaryTree 5H`, `BinaryTree 6H`, and `Remove 4H`. For `BinaryTree 6H`, $ASketch_{EvalSub}$ goes from taking 12.1 minutes to find a solution with one thread to a more reasonable 2.8 minutes with eight threads.

Multi-threaded approaches are limited by the system that is running the program, and this factor should come into play when determining how many threads to run. However, even generating two threads provides a reduction for `BinaryTree 6H` from just over 12 minutes to just under 7 minutes. Since exploring the ranges can be done in parallel without causing thread conflicts which erode the benefits of a parallel approach, a multi-threaded implementation of $ASketch_{Eval}$ is recommended.

6.4 Exploring Test Order

There is a simple optimization used by all $ASketch_{Eval}$ techniques that is irrelevant to the variability of Alloy and the order of candidate exploration: $ASketch_{Eval}$ explores the candidate solutions by looping over tests, and exits this loop as soon as a candidate fails a test. As a result, the order in which tests are explored has some impact on the time to the first solution. Consider a test order in which the first two tests eliminates 75% of the candidates compared to a test order in which the first ten tests are needed to eliminate

75% of the candidates. The first test order would result in less work per candidate, the main feature separating the base $ASketch_{Eval}$'s performance from the optimized versions. To explore just how much of an impact the test order has on $ASketch_{Eval}$'s run time, we conducted an experimental evaluation of random test orders to answer the following questions: Does the order of tests have a noticeable impact of the runtime? If so, what factors define a good test order? And can these factors form a set of guidelines for test generation for sketching Alloy models?

6.4.1 Random Test Order Experiment

To gather insight into what makes a test order "good", we set up a random test order experiment across all 24 model variants. Each run permutes the test order with the only restriction being that the new order does not match any previous order. Table 6.2 shows the results of exploring 100 random test orders using $ASketch_{EvalSub}$. Columns 1 and 2 are used to show which model variant is being considered. Then, column 3 shows the minimum runtime, column 4 shows the maximum runtime and column 5 shows the average runtime across all 100 runs. All times reported in the table are in seconds.

From these results, it is clear that some test orders are *better* than others, as seen by `BinaryTree 6H` where the minimum run time is faster than the maximum run time by a factor of 4.3, which equates to the minimum order finishing 37 minutes faster than the maximum order. Across all the model variants, the minimum run time is anywhere from 2 to 4.5 times faster

Table 6.2: Results for 100 Random Test Order Execution using $ASketch_{EvalSub}$.

Model		**Min (s)**	**Max (s)**	**Avg. (s)**
LinkedList	1H	0.00	0.01	0.00
	2H	0.00	0.01	0.00
	3H	0.11	0.37	0.26
	4H	0.19	0.63	0.36
	5H	0.39	0.76	0.56
BinaryTree	1H	0.00	0.03	0.01
	2H	0.20	0.71	0.36
	3H	0.19	0.75	0.35
	4H	1.68	4.99	2.78
	5H	7.58	28.52	13.03
	6H	670.73	2904.94	1193.48
Contains	1H	0.02	0.05	0.03
	2H	0.02	0.04	0.03
	3H	0.07	0.27	0.13
Remove	1H	0.08	0.31	0.16
	2H	0.17	0.96	0.40
	3H	0.25	1.17	0.53
	4H	6.70	13.77	8.72
Dijkstra	1H	0.00	0.00	0.00
	2H	0.01	0.07	0.02
	3H	0.01	0.06	0.03
	4H	0.01	0.07	0.03
	5H	0.03	0.07	0.04
	6H	0.40	0.78	0.55

than the maximum run time. The order of tests having such an impact on the run time is similar as to why the equivalence class optimizations $ASketch_{EvalExp}$ and $ASketch_{EvalSub}$ can make substantial decreases in run time: the more candidates eliminated early, the less overall work needs to be done at a per candidate level. When exploring potentially billions of candidates, which `BinaryTree 6H` and `Remove 4H` can require, catching a majority of the candidates early is significant. In detail, for the best test order from the random runs over `BinaryTree 6H`, the first three tests eliminate 98.5% of all the candidate solutions. On the other hand, for the worst test order, the first three tests eliminate just 7.7% of the candidates, and nine tests are required to eliminate over 98.5% of the candidate solutions, which as we have seen, led to a difference of 37 minutes in their run times.

6.4.2 Guidelines for a Sketching-Based Test Suite

Since the test order does impact the run-time, we want to extract some general lessons from stepping over the random test order results, in combination with lessons learned from generating our own test suites for sketching. To form our initial test suites, we started with a base of two tests. Then, we used our $ASketch_{EvalSub}$ technique to search for the first incorrect solution and used the solution to produce a counterexample which we turned into a test. This process was repeated until only valid solutions were found. Through our analysis of the random test orders, we gained additional insight into why our initial test generation process produced certain tests and did not produce other tests. In total, we present three guidelines to follow when generating and ordering a test suite for sketching purposes.

1. Explore different valid and invalid behaviors first. Our first and most important guideline is to focus on *formula* level differences at the beginning of a test order. When looking out our best performing test orders, the initial tests explore a range of different ways that the formula being sketched

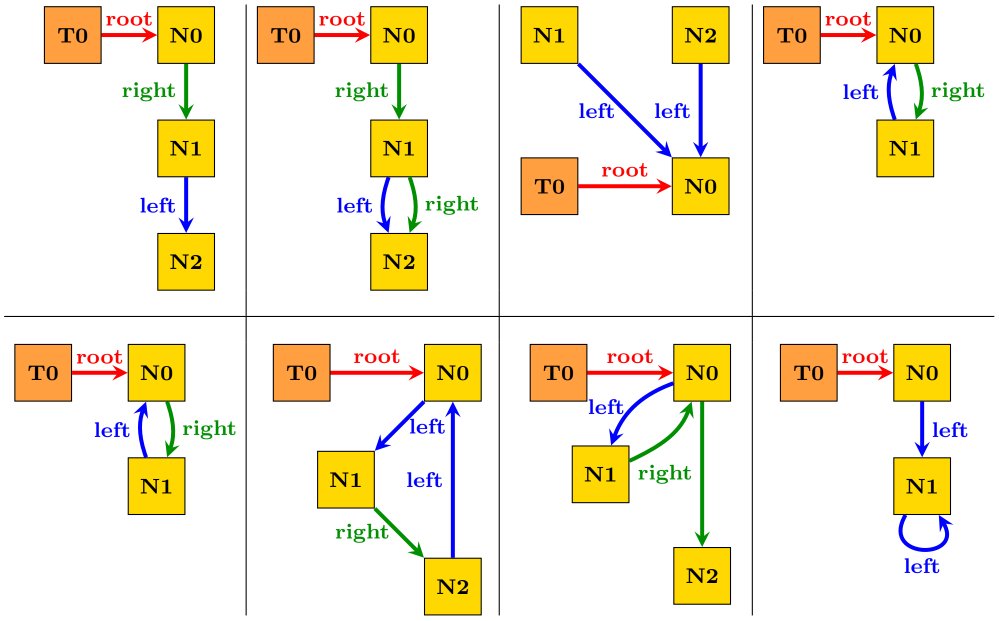


Figure 6.1: First four tests from the best test order (top) and first four test from worst order (bottom) for the binary tree variant with 6 holes.

could be true or false. For instance, for our `BinaryTree` model, there are three basic criteria for a binary tree to be well-formed: (1) no node in the tree is reachable from itself, (2) any node in the tree has one parent, and (3) no node in the tree left and right fields point to the same node. Figure 6.1 reflects the first 4 tests from the best random order over `BinaryTree 6H` on top, and first 4 tests from the worst random order on bottom.

As we can see in Figure 6.1, the fastest random test order covers: (1) the formula being true, (2) the formula being false because a node's left and right relations point to the same node, (3) the formula being false because a node has multiple parents and (4) the formula being false because a node is reachable

Table 6.3: Characteristics of the first 5 tests in the best and worst test orders.

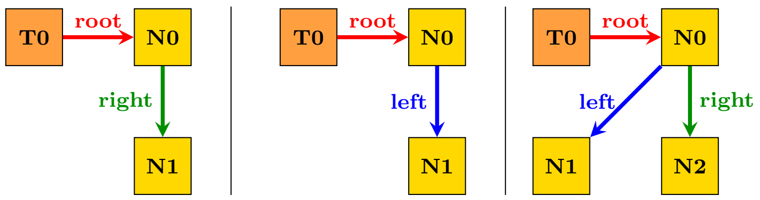
Model	**Ord**	**% Cand Eliminated**					**Formula Coverage**				
		1	2	3	4	5	1	2	3	4	5
LinkedList 5H	B	60.0	94.6	96.4	98.6	99.9	13.6	50.0	72.7	77.2	90.9
	W	60.4	61.4	65.5	93.8	96.4	30.3	33.3	33.4	54.5	54.5
BinaryTree 6H	B	91.1	94.2	98.5	98.9	99.6	41.7	58.3	79.1	95.8	100.0
	W	4.9	7.0	7.7	16.5	20.1	41.7	66.7	66.7	66.7	83.3
Contains 3H	B	83.8	98.1	98.1	98.1	100.0	40.0	100.0	100.0	100.0	100.0
	W	17.1	19.0	19.0	20.0	20.0	20.0	20.0	20.0	60.0	60.0
Remove 4H	B	84.3	92.4	96.2	99.5	99.6	28.6	42.8	57.1	85.7	85.7
	W	12.5	19.5	22.7	74.6	98.5	42.8	57.1	57.1	68.2	85.7
Dijkstra 6H	B	82.2	91.8	98.9	99.9	99.9	37.5	68.7	75.0	93.7	100.0
	W	17.8	26.4	26.9	30.4	36.3	25.0	68.7	68.7	75.0	81.2

from itself. Furthermore, nodes are both reachable from the tree and not. However, the first 4 tests from the worst order all focus on one main failing attribute: a node reachable from itself. Additionally, due to a self loop, the fourth test also captures a node having multiple parents and unlike our best ordering, all nodes are in the tree.

To expand on this observation, Table 6.3 dives into the behavior of the first 5 tests for the best (B) and worst (W) performing test orders. Column 1 highlights the model variant under consideration: we will look at the largest model variant for each base model. Column 2 shows which test order is being explored. Columns 3 through 7 show the percentage of candidate solutions eliminated by the first x tests in increments, i.e. the first 1 test, the first 2 tests, etc. Columns 8 through 12 show the formula level coverage of the first x tests in the same increments.

Table 6.3 show the worst test orders have a period of stagnant formula coverage at the beginning of their test suites. In contrast, the best test orders are constantly improving their formula coverage. When we compare this to the percentage of candidates being eliminated, we can see the direct impact that this stagnant coverage period has on our ability to eliminate candidates. While all of our tests will eliminate some candidates, having stagnant formula coverage means each additional test provides the same coverage as some previous test, thus reducing a test's ability to eliminate unique candidate solutions compared to the preceding tests. To see this in detail, we can look at the worst order for `LinkedList 5H`. Test one through three have a period of nearly stagnant coverage, which correlates to minute increases in the percentage of candidates being eliminated. However, the jump in coverage with test four translate to a big jump in the percentage of candidates eliminate. Of note, the best test order does not always start with or have the highest formula coverage, but the best test order is constantly improving the formula coverage in the beginning.

However, an important caveat to this guideline is that we do not want to ignore the intricacies of how these passing and failing behaviors occur when *generating* a test suite. Below are 3 valid valuations of our `BinaryTree` model from our full test suite:



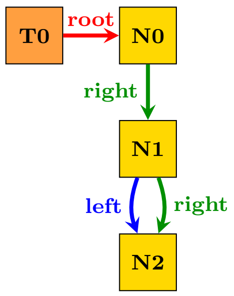
These 3 tests eliminate unique candidates with respect to each other; therefore, all three are important when it comes to sketching `BinaryTree`. However, there is significant overlap in the candidate solutions these tests eliminate. Accordingly, we would not want to place these three tests next to each other in our test order.

Based on Table 6.3 and graphical inspections of good and bad test orders, we can see the importance of focusing on exploring diversity at a formula level viewpoint in the beginning of the test order. After which, we can narrow in on the different ways these behaviors can occur, such as one tree being valid because of the `left` relation, a different tree being valid because of the `right` relation and a different tree being valid because of *both* the `left` and `right` relation.

2. Note any concrete formulations of your sketch. Any part of the to-be-sketched formula that is concretely provide can have an impact on the value of a test. To form our test suites, we used the largest model variant (the model variant with the most holes) to produce the test suite. Therefore, some of the tests will reason over concrete portions of our smaller model variants, which do not have a fully abstracted formula. Consider the partial model for our variant `BinaryTree 4H`:

```
one sig BinaryTree { root: lone Node }
sig Node { left, right: lone Node }
pred IsTree{
   all n : Node | n in BinaryTree.root.*(left + right) => {
      n \CO\ \E\ and \UO\ \E\ and no n.left & n.right
   }
}
```

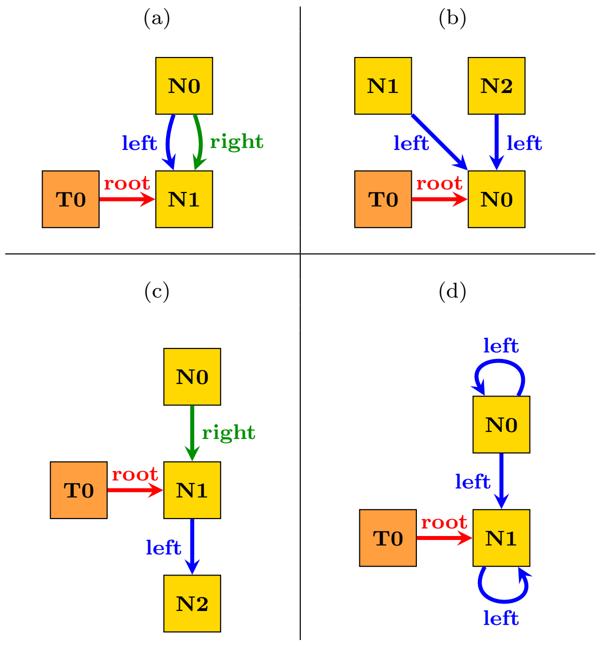
For this model variant, two of the properties needed to produce a well-formed tree are abstracted away into a sketch, while one property, no node in the tree has the same left and right relation, is concretely specified. When we look at the slowest orders for `BinaryTree 4H`, we find that the following invalid test is in the beginning of test orders:



This valuation is in place to help sketch `"no n.left & n.right"`, which is explicitly specified in `BinaryTree 4H`. As a result, the test depicted above eliminates 0 candidates – the concrete portions of the formula ensure that every candidate solution will correctly evaluate to `false` for this test valuation.

We can further see this behavior if we look into why the empty tree is never produced as a test for our `BinaryTree` model. Typically, the empty instance is an interesting corner case to consider; however, if we add the empty tree to our sketching test suite, we will find that the test eliminates 0 candidates across all `BinaryTree` model variants. If we take a closer look out our sketch for `BinaryTree`, we can see the sketch is embedded in a universally quantified formula (`"all n : Node {...}"`). As a result, over the empty tree, all candidate solutions will simply reduce to the vacuous result of this quantified formula. In other words, the substitutions we make into the holes has zero impact on whether a candidate solution will pass this test, resulting in zero candidate solutions being eliminated. Therefore, this is an important guideline to take into account when generating a test suite for sketching, as a test which eliminates no candidates will only serve to slow down the runtime.

3. Remember, in Alloy, any behavior you have not ruled out is valid. Our last guideline is really a general reminder that in Alloy unless you rule out a behavior the Alloy model will allow it, which should factor into the characteristics you capture in the valuations for your test suite. Consider the case of generating tests for our `BinaryTree` model for sketching. We should take into account our various signature and relation multiplicity constraints. However, we do not have a "connected constraint" requiring all `Node` atoms to be in the binary tree. Therefore, it is valid for nodes not to be in the tree, and this behavior can have an impact on the validity of the tree itself. Below we depict 4 different valuations which come from our `BinaryTree` full test suite, where valuations(a) and (c) are labeled valid and valuations (b) and (d) are labeled invalid:



These test valuations show that (a) it is possible to have invalid disconnected nodes as a parent to a node in a valid tree; (b) valid disconnected nodes can lead to an invalid valuation; (c) valid disconnected nodes can still lead to valid valuations; and (d) invalid disconnected nodes can still lead to invalid valuations. These varying ranges of behaviors all serve to eliminate candidate solutions and are all based on exploring the impact of nodes disconnected from the tree.

Whether the test order was fast or slow, at about the half way mark through the test order, around 98% of the candidate solutions have been eliminated. The second half of the test suite serves to finalize the sketching behavior. However, there is a notable difference between how quickly the faster test orders eliminate the first 98% compared to how quickly the slower test orders eliminate the first 98% of candidates. Our slower test orders always contain a period in the beginning in which minimal gains are made to eliminate candidates, thus increasing the amount of work done, on average, to rule out incorrect candidates. In the end, our first guideline – cover the unique high-level properties at which you envision the to-be-sketched formula to be true and false – is the most important factor for an efficient test order for sketching. However, adhering to our second and third guidelines remains important when generating the test suites as ignoring these guidelines can result in tests with poor ability to eliminate candidates on their own.

6.5 Summary

Currently, $ASketch_{EvalSub}$ is the recommended methodology for sketching Alloy models. In addition to an enumeration based technique, [75] presents $ASketch_{Solve}$, a solving based approach to sketching Alloy models. $ASketch_{Solve}$ encodes the model under sketch, the test cases and the candidate fragments into one aggregate Alloy model, and asks the Alloy Analyzer to solve for a solution to the sketch using its SAT-based backend. The results in [75] show that the more advanced $ASketch_{Solve}$ technique using a symbolic approach performs worse than the simpler $ASketch_{Eval}$ technique based on enumeration (when accompanied with good optimizations in $ASketch_{EvalSub}$), with the notable exception of `BinaryTree 6H`, the case that is the motivation behind our efforts to improve $ASketch_{Eval}$. Accordingly, this chapter focuses on two different avenues to improving $ASketch_{Eval}$'s runtime. Due to $ASketch_{Eval}$'s enumeration-based approach, a multi-threaded implementation is a well-suited method for improving $ASketch_{Eval}$'s runtime, as our results show. In addition, our random test order experiments reveal that even the order of tests can drastically impact the run-time of a sketch. To reduce the likelihood of a user supplying a bad test order, we introduce a series of sketching-based test suite guidelines, directed at both the order of the tests and the generation of the tests.

Chapter 7

Formula-Based Sketching Of Alloy Models

7.1 Overview

The work outlined in previous chapters has focused on using traditional testing paradigms – automated test generation (Chapter 4) and program synthesis through sketching (Chapter 5) – to help develop a stronger validation environment for Alloy. Prior to the introduction of AUnit as a testing framework, end users of Alloy performed ad-hoc measures to validate their models within the Alloy Analyzer toolset using two common approaches. One, instance enumeration where users run a command in Alloy and visually inspect the instances displayed by the Alloy Analyzer for their correctness. Two, assertion checks, where the user looks for the existence of expected logical relationship between formulas, and uses an assertion to check that the relationship holds.

Our initial approach to sketching Alloy models is rooted in the development of AUnit. Specifically, *ASketch* is based on the insight that test valuations enable the user to outline the expected behavior of an Alloy model. Using this expected behavior, *ASketch* can iterate over all possible solutions to a sketch, eliminate candidates which violate that behavior, and report solution(s) to the user which match the behavior, i.e. pass all the tests. While test valuations are an important input to *ASketch*, we can utilize existing validation practices of Alloy users for *ASketch* inputs.

In particular, we introduce $ASketch_{Equiv}$, a formula-based sketching approach which is centered around the logical equivalence between an input formula and the to-be-sketched formula. A common assertion used for validation is checking for the logical equivalence between two formulas. By definition, a formula intended to be equivalent to another formula outlines the expected behavior of the second formula. As a result, by taking an equivalent formula as input, $ASketch_{Equiv}$ does not require a user-provided test suite. Instead, $ASketch_{Equiv}$ uses $AGen_{WB}$ to produce a test suite based on the equivalent formula, orders the test suite based coverage information, and lastly utilizes an $ASketch_{Eval}$ back-end to solve the sketch.

This chapter makes the following contributions:

Formula-based sketching: We introduce $ASketch_{Equiv}$, a technique to sketch Alloy models based on a user provided equivalent formula instead of user provided test suite. The technique uses a modified version of $AGen_{WB}$ to generate a test suite and leverage $ASketch_{EvalSub}$ to solve the sketch.

Embodiment: Our implementation for $ASketch_{Equiv}$ is based on both the embodiment of $AGen_{WB}$ and $ASketch_{Eval}$. Furthermore, our toolset provides a concrete implementation of the sketching-oriented test suite guidelines from the results of our random test order experiment (section 6.4).

```
module list.v.3
one sig List { header: lone Node }
sig Node { link: lone Node }

pred AcyclicSome {
   no List.header or some n : List.header.*link | no n.^link
}

pred AcyclicOther {
   \Q\ n : List.header.*link | \E\ \CO\ \E\
}

check { AcyclicOther <=> AcyclicSome }
```

Figure 7.1: A singly-linked list model with one specified acyclic predicate and one sketch predicate.

Experiments: We perform an evaluation of $ASketch_{Equiv}$ over a suite of Alloy models that have two different formulations of the same property. Our experimental results show that despite structural differences in the formulas, $AGen_{WB}$ test suites in combination with with the embodiment of our test order guidelines are able to sketch Alloy models effectively.

7.2 Example

To explore sketching through equivalence, we will step over a model of a singly-linked list with two different representations of an acyclic property – one a fully specified predicate and the other a to-be-sketched predicate.

Our model is introduced by the keyword `module` which names the model, and enables the model to be imported into other models. Next, two signatures paragraphs are declared (`sig List` and `sig Node`). Signatures introduce a set of *atoms*, e.g. our model has two sets: one named `List` and a separate set named `Node`. Within a signature paragraph, 0 or more relations can be introduced. Relations can introduce relationships between atoms within the same set – e.g. `link` establishes a binary relation relating a `Node` atom to zero or one `Node` atom, or relations can relate atoms over different sets – e.g. `header` ensures each list's header is either the empty set or a Node atom. Signatures and relations can have multiplicity constraints such as "`one sig List`" which requires the List set to be comprised of *exactly* one `List` atom.

Next, our model introduces two predicate (`pred`) paragraphs. The first, `AcyclicSome`, uses *disjunction* (`or`) and *existential* quantification (`some`) to define acyclic. The domain of the quantified formula is the set of `Node` atoms in the list ("`List.header.*link`"). This is achieved by using set join("`.`") and *reflexive* transitive closure ("`*`") to capture the set of `Node` atoms reachable from zero or more traversals down the List's `header`'s `link` relation. Using reflexive transitive closure instead of transitive closure ("`^`") ensures that this set includes the List's `header`. Any `Node` atom in this domain, `n`, is evaluated to see if `n` has no `link` relation. In other words, if there is a null-terminated node in the list, there cannot be a cycle in the list. In order for the existentially quantified formula to be *true*, then at least one element in the domain must satisfy this constraint. As a result, an empty list is incorrectly ruled out. Therefore, we include the disjunction to account for the presence of either an empty list (`no List.header`) *or* some node in the list is null-terminated.

The second predicate, `AcyclicOther`, is looking for another formulation of acyclic. The intent is to discover a representation that does not need the extra disjunction constraints, and instead can reason over a single quantified formula. The predicate is primarily comprised of a series of holes: one quantifier hole (\`Q`\), 2 expression holes (\`E`\), and one compare operator hole (\`CO`\). The only part of the formula given concretely is the declaration of the variable `n` and its domain `List.header.*link`, as variable declarations are not currently supported by sketching.

The last part of our model is a `check` command, which asserts that our `AcyclicSome` predicate should be true if and only if our sketched `AcyclicOther` predicate is true. In other words, this check is looking for an equivalence relationship between our two predicates. If this relationship does not hold, the Alloy Analyzer will produce a counterexample, a valuation in which a difference in behavior between the two predicates can be seen.

With fragments of size 5, 57, 4, and 57 for \`Q`\, \`E`\, \`CO`\, and \`E`\ respectively, $ASketch_{Equiv}$ searches through a potential state space of 64,980 solutions to find the following sketch for `AcyclicOther`:

```
all n : List.header.*link | n.^link in (Node - n)
```

The solution uses universal quantification (`all`) to state that for all nodes in the list, the set of nodes reachable from the list (`n.^link`) is a subset of all nodes minus itself (`Node - n`). $ASketch_{Equiv}$ finds this solution in 0.08 seconds, after taking 0.15 seconds to build a starting test suite consisting of 5 tests, and will generate 1 additional test while sketching.

7.3 Sketching by Equivalence

In this section, we present our $ASketch_{Equiv}$ technique, which utilizes our previous *ASketch* framework (Chapter 5) and our automated input generation technique $AGen_{WB}$ (section 4.3.2). $ASketch_{Equiv}$ is a formula based sketching approach, which unlike $ASketch_{Eval}$, does not require the user to provide a test suite. Instead, the user provides an Alloy formula which is equivalent to the formula the user wants to sketch. This formula does not need to have the same structure as the to-be-sketched formula and does not need to be formally presented within the model as a predicate or assertion. Given an Alloy model with a sketch and an equivalent formula, $ASketch_{Equiv}$ proceeds in two steps. First, a test suite is generated and re-ordered based on the guidelines from section 6.4. Second, the sketch is solved, using a modified $ASketch_{EvalSub}$ approach.

7.3.1 Building and Ordering a Test Suite

In contrast to our prior sketching approaches ([75]), a test suite is not a user provided input. Recall, for $ASketch_{Eval}$, the test suite facilitated the sketching process. Specifically, $ASketch_{Eval}$ checks that candidate models pass all tests and additionally, the tests enable our suite of optimizations shown to be effective at reducing the run time (section 5.4). In order to make direct use of $ASketch_{EvalSub}$, the current recommendation for sketching Alloy models, as the backbone of $ASketch_{Equiv}$, we need a test suite. Since the user does provide an equivalent formula, we can leverage prior work which established a methodology for automatically generating test suites in Alloy (Chapter 4).

Generating the test suite: Specifically, we will use a modified version of $AGen_{WB}$, coverage-directed input generation. There are a few notable differences with the approach presented in section 4.3.2. First, we will look to generate a test suite which achieves maximum coverage of the equivalent predicate. Additionally, there is no need for any human intervention as an oracle of the tests because our equivalent predicate will act as a oracle for the generated test suite. Lastly, when generating tests for coverage only, we are interested in covering criteria that violate the facts of the model, as our intent is to provide a small, but robust test suite to catch any bugs in the model, including those in the facts. However, with sketching, the model is taken as is, meaning the facts and any signature or relation constraints are assumed to be valid formulations. Therefore, we are not interested in generating tests over coverage requirements which violate the facts of the model.

When it comes to generating the test suite, we can re-use the structure of algorithm 2 presented in section 4.3.2, as our changes to $AGen_{WB}$ are in the perimeter. Specifically the call to `M.extractRequirements()` will now reflect the changed scope of coverage criteria we are looking for. Next, we will drop the first `if` statement which checks if the instance produced by the targeting constraint is satisfiable. Instead, we immediately treat the targeted criteria as infeasible if running the empty command is unsatisfiable. Lastly, our test cases can be immediately labeled valid and invalid based on their valuation over the equivalent predicate. This is already done by the algorithm, but $AGen_{WB}$ requires a human oracle to confirm that this invalid or valid distinction is the expected behavior. In our case, we can assume that the result based on the equivalent predicate is accurate.

Ordering the test suite: In section 6.4, experimental results show that the order of tests has an impact on the sketching time; therefore, we want to apply these guidelines to ensure we use an efficient test ordering. While $AGen_{WB}$ produces test suites with maximum coverage, there is not a guarantee that the tests are in an order which achieves maximum coverage at the fastest rate. Additionally, we are not concerned with overall model coverage, which would factor in signature, relation, expression and formula coverage as equal contributions. Instead, we are concerned with exploring a broad range of passing and failing behavior over our equivalent formula, before narrowing in on specifics, such as the size of expressions when the formula containing the expression is true. Therefore, our test suites will be rearranged to achieve complete formula coverage first, then expression coverage, and lastly ordering based on signature and relation coverage.

7.3.2 Completing the Sketch

With a test suite generated and reordered, we can now use various aspects of our *ASketch* framework to complete the sketch. First, we still have the problem of $ASketch_{Gen}$, in which we will need to produce a list of candidates fragments for each hole in the model. Once again, we can use the Alloy grammar to outline the fragments for all operator holes i.e. `\UO\`, `\LO\`. In [75], we propose a methodology for automatically building up fragments for expression holes, which was used previously to generate the expression fragments for the evaluations done in Chapter 5 and Chapter 6. $ASketch_{Equiv}$ will use the dynamic pruning $ASketch_{Gen}$ approach to generate the needed expression fragments, which makes smaller fragment lists but can take minutes to run. To solve the sketch, we will use the optimized $ASketch_{EvalSub}$, which currently outperforms all other Alloy sketching algorithms.

Algorithm 7 outlines the overall framework of $ASketch_{Equiv}$, combining all of the different parts together to complete the sketch. First, we build the test suite, using the modified $AGen_{WB}$ technique outlined in the previous section (`buildTestSuite`). Then, we re-order the test suite, as needed (`orderTests`). Next, we can start sketching following $ASketch_{EvalSub}$'s algorithm in which we use a memoized `values` map to form equivalence classes over the candidates based on their expression and subformula evaluations over each test case. The `values` map is leveraged to help build a representation for a candidate solution where a second map, `cache`, is used to determine if this representation has been encountered before. If this representation is encoun-

Algorithm 7: *ASketch*$_{Equiv}$.

```
Input: ASketch model (M), equivalent formula (equiv_f), fragments for
       each hole (D)
Output: candidate solution
State: values: (Test,String) ↦ String // caches computed
  values ← {} // empty map
  cache ← {} // empty map (Test,String) ↦ Boolean
  buildTestSuite(M)
  orderTests(M, ts, equiv_f)
  foreach cand in buildCandidates(M, D) do
      addTest ← false;
      foreach t in ts do
          rep ← ””
          // Perform subformula optimization
          foreach subf in subformulas(M) do
              rep += memoSubFormula(t, subf)
          foreach hole h not in a subformula do
              rep += cand.getValueForHole(h)
          if !cache.containsKey(t, rep) then
              if !eval(t, cand) then
                  cache[(t, rep)] = false
                  break
              cache[(t, rep)] = true
          else if !cache[(t, rep)] then  break
          if last test case then
              val ← checkCommand(“cand <=> equiv_f”)
              // if check unsatisfiable, then no counterexample found,
                formulas are equivalent
              if val.unsatisfiable then return cand
              else  addTest ← true
      if addTest then
          // check if instance should be allowed by the formula
          if val.eval(equiv_f) then
              ts.add(new Test(val, valid))
          else
              ts.add(new Test(val, invalid))
          values.put(ts.last, new String ↦ Boolean)
          cache.put(ts.last, new String ↦ Boolean)
```

tered for the first time, the representation is evaluated across the test case; otherwise, the previous encounter with this representation is re-used.

However, when a candidate solution passes all tests, unlike $ASketch_{Eval}$, $ASketch_{Equiv}$ has the ability to know if a candidate solution matches the end-user's expectation. Specifically, we can ask Alloy if the two formulas are equivalent, using a `check` command of the form:

```
check { candidateSolution[] <=> equivalentFormula[] }
```

If this check *fails*, then we know that we have found an incorrect solution to the sketch. We can adjust our test suite by taking the counterexample produced by the sketch and turning it into a new test case. The valuation will be the instance captured by the counterexample and the command will be valid if the counterexample is an instance allowed by the equivalent predicate, and labeled invalid otherwise. With the new test added to the suite, we can *resume* sketching – all of our previous eliminated candidates are still incorrect and the `cache` and `values` mappings still reflect accurate information to continue re-using as required by the optimizations. Both mappings `values` and `cache` will be extended to start collecting results for the new test case. Once this equivalence check passes, the result is presented to the end user. This notion introduces the spirit of iterative sketching, addressing the case where the sketch remains the same but a test has been added to the suite.

While this is an easily built in functionality, our goal is to reduce the need to generate any new tests while sketching. Section 6.4 highlights the test orders role in *ASketch* execution time, with the first handful of tests having the largest impact. Therefore, we want to go into the sketching effort with an initial set of tests where the first few tests together eliminate a majority of the incorrect candidate solutions. Should our $AGen_{WB}$ technique not generate some of the needed tests, we want to keep that to a minimal amount of narrow corner cases. In particular, we do not want to miss any core features of our sketch such that we end up relying on our tests generated while sketching to eliminate a significant portion of the candidates, which could significantly impact the runtime.

7.4 Evaluation

7.4.1 Subject Models

Our evaluation is based on a suite of 7 core Alloy models. First, we have a series of data structure based models, starting with the `LinkedList` from our running example (Figure 7.1). Additionally, we use a model of a `BinaryTree` which focuses on the acyclic property of a binary tree, `Contains` reasons over whether or not a given element is in a list, and `Remove` looks at whether a given item has been removed from a list. Then, we have three algorithm based models: `Dijkstra` that reasons over the deadlock property that Dijkstra's mutex ordering algorithm should prevent, `StronglyConnect` which enforces that a graph is strongly connected, and `GphColor` that looks to color a graph such that no connected nodes are the same color.

Each Alloy model has two versions of a predicate that are intended to be equivalent. To confirm, we used Alloy to *check* for equivalence over the scope used in sketching. Additionally, for every model, the two predicate versions have different formula structures. For each model, one version of the predicate is set as the "equivalent" predicate, and the other is turned into a sketch, starting one hole at a time. For instance, for our example singly-linked list model, the to-be-sketch predicate would be formed into a series of sketches as follows:

```
\Q\ n: List.header.*link | n !in n.^link
\Q\ n: List.header.*link | \E\ !in n.^link
\Q\ n: List.header.*link | \E\ \CO\ n.^link
\Q\ n: List.header.*link | \E\ \CO\ \E\
```

Our expression fragments were produced uses the dynamic pruning approach to $ASketch_{Gen}$. The experiments were run on an Intel Core i7-4790K Quad-Core 4.0 GHz on Windows 7.

7.4.2 $ASketch_{Equiv}$ Performance

Table 7.1 captures various performance metrics for our $ASketch_{Equiv}$ technique. The first column outlines which model variant of a subject model is being used. Column 2 reflects the time taken to produce the appropriate test suites, this includes the time to generate the test suites, as well as the time to re-order the test suite, if needed. Columns 3 and 4 reflect the size of the test suite at the start and end of $ASketch_{Equiv}$'s execution: the test suite may grow in size if the $AGen_{WB}$ produced test suite is not adequate. The last column captures the sketching time: the time from starting the sketch to the time the first solution is found. This time includes the time to generate a new test, if needed. All solutions were checked for *correctness* by using Alloy to check that the solution formula is equivalent to our intended formula.

The time taken to generate our test suites is small, with most model variants taking less than a second to produce the test suites. Additionally, based on the time taken to complete the sketch, our $ASketch_{EvalSub}$ remains effective at sketching Alloy models. In particular, most of the sketching times are under a second and almost all are under 10 seconds. Our `BinaryTree 6H` model variant retained most of the complexity from our prior sketching work, resulting in the longest run time out of all model variants.

As a whole, the $AGen_{WB}$ based test suites were sufficient to sketch a majority of the sketches. At most, 4 additional tests are generated, as seen in `BinaryTree 6H`, and occasionally one or two tests were generated for the larger model variants. For all model, the smaller model variants do not need to generate any additional tests, which is expected due to the smaller solution space for the sketching problem. An interesting occurrence is the generation of additional tests for the `Remove 4H` model variant but not the `Remove 5H` model variant. A similar situation occurs with the `StronglyConnect`, in which an additional test is produced for model variants `4H` and `5H` but not `6H`. Generally, the model variant with the most holes is the "hardest" model to sketch, meaning that typically the entire formula structure has been abstracted away, producing the largest state space to reason over. However, due to the expressive nature of Alloy, these large model variants can have multiple equivalent

Table 7.1: $ASketch_{Equiv}$ performance. Times are in seconds.

Model		T_{create}	T_{suite} Size		T_{sketch}
			Start	End	
LinkedList	1H	0.17	5	5	0.00
	2H	0.18	5	5	0.00
	3H	0.19	5	5	0.00
	4H	0.15	5	6	0.22
BinaryTree	1H	4.32	13	13	0.00
	2H	4.92	13	13	0.18
	3H	5.08	13	13	0.13
	4H	4.69	13	15	1.67
	5H	4.75	13	15	6.06
	6H	4.77	13	17	704
Contains	1H	0.19	5	5	0.00
	2H	0.23	5	5	0.00
	3H	0.16	5	5	0.01
	4H	0.18	5	5	1.20
	5H	0.16	5	5	0.04
Remove	1H	0.82	8	8	0.00
	2H	0.95	8	8	0.42
	3H	0.82	8	8	0.24
	4H	0.73	8	10	1.55
	5H	0.86	8	8	6.27
Dijkstra	1H	0.34	6	6	0.00
	2H	0.37	6	6	0.00
	3H	0.34	6	6	0.00
	4H	0.30	6	6	0.00
	5H	0.35	6	6	0.00
	6H	0.27	6	6	0.44
GphColor	1H	0.93	4	4	0.00
	2H	0.89	4	4	0.00
	3H	0.97	4	4	0.00
	4H	0.78	4	6	1.12
StronglyConnect	1H	0.70	9	9	0.00
	2H	0.68	9	9	0.00
	3H	0.72	9	9	0.23
	4H	0.69	9	10	0.20
	5H	0.67	9	10	0.17
	6H	0.69	9	9	0.28

solutions, which increases the likelihood of finding a solution early in the enumeration process. Meanwhile, the model variants with most but not all of the formula abstracted to a sketch have less overall solutions to be found. In these cases, the solution required looking over a number of expression combinations, which facilitated the need for additional tests. However, while the problem to be sketched has much narrower criteria to meet, the solution space is small and sketches are completed quickly regardless.

Some of the equivalent formula pairs are similar in nature to each other, such as `BinaryTree` and `Dijkstra`. However, other formulas are markedly different, such as our linked list example which uses universal quantification and set-based compare operators for one formula, and disjunction, existential quantification and unary operators for the other formula. Another model that reasons over two markedly different structures is `StronglyConnect`, which works with the following formula pairs:

```
1. no iden & adj and all n : Node | Node - n in n.adj
2. adj = {Node -> Node} - iden
```

The first formula first asserts that there is no intersection between the set "`iden`" and the relation "`adj`" (`iden` is a keyword for the set containing the identity of all atoms in the model i.e. for an atom `n` its identity value is `n->n`). Then, the first formula uses universal quantification to state that every node's `adj` relation should be a map to all nodes except itself, using set inclusion (`in`). On the other hand, the second equation simply uses set equality (`=`) to state that the `adj` relation should be populated with the Node $\times$ Node cross product, minus any mapping which is also in `iden`.

Despite these structural differences in the formulas, across the board, the $AGen_{WB}$ test suites served as a strong starting base across all the model variants. Although not always adequate, the number of candidate solutions eliminated by any newly generated test was minimal, numbering from 0.08% to 0.68% of the total number of candidates eliminated across all the model variants.

To see why tests were added during sketching, we can look at our running example, the singly-linked list model. The $AGen_{WB}$ test suite was adequate for all sketches until the largest model variant `LinkedList 4H`, where one test was added. Figure 7.2 shows two invalid valuations that have the same coverage. The valuation depicted in (a) is generated by $AGen_{WB}$ while valuation(b) is the test added to the suite during $ASketch_{Equiv}$'s execution. Recall our `AcyclicSome` equivalent formula is the following:

```
no List.header or some n : List.header.*link | no n.link
```

The valuations in Figure 7.2 are both related to the quantified formula criteria "`|d| > 1 and b = false for all`." However, these two valuations capture two different ways in which all `Node` atoms in the list can violate the body (`no n.link`): (1) a node in the list has a self loop and (2) a node in the list points to an earlier node in the list. As a result, $AGen_{WB}$ will produce one of these two valuations, but not both. While both tests are necessary to

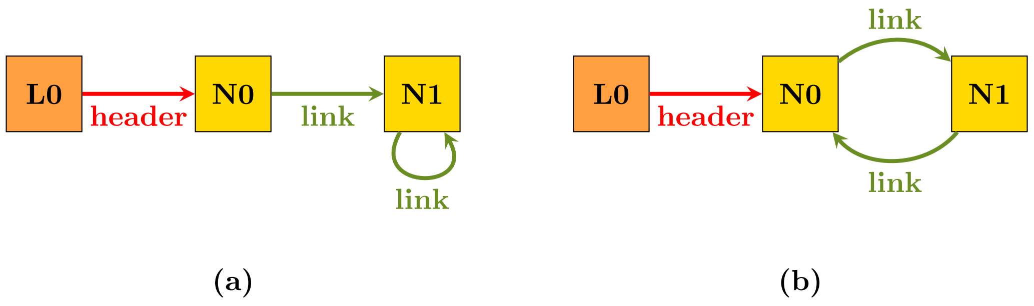


Figure 7.2: Two invalid valuations with the same coverage.

sketch `AcyclicOther` *correctly*, the two tests have significant overlap in the candidates they eliminate. Specifically, the addition of the test in Figure 7.2(b) is used to eliminate just 0.68% of the candidate solutions encountered. Meanwhile, the test in 7.2(a) is the second test in the order, and together the first two tests eliminate 65.18% of the candidate solutions while the first three eliminate 85.18% of the candidate solutions. Therefore, while $AGen_{WB}$ can currently miss some of the intricate details on how different passing or failing behaviors can occur, since any added test serves to eliminate minor portions of the candidate solutions, our $AGen_{WB}$ test suites do not miss any important functionality of the formula being sketched.

7.5 Summary

The $ASketch_{Eval}$ approach to sketching works directly with AUnit, by requiring a collection of user provided test valuations. On the other hand, $ASketch_{Equiv}$ uses $AGen_{WB}$ to enable the sketching of Alloy models through equivalence. Rather than having the end user develop a test suite, $ASketch_{Equiv}$ makes use of an existing ad-hoc Alloy validation technique: the use of assertions to check for the logical relationship between two formulas. While the equivalent formula can take any form, $AGen_{WB}$ has proven to generate strong test suites for sketching, typically capturing most, if not all, of the to-be-sketched problem. $AGen_{WB}$ has already proven effective at finding real world bugs; however, based on the few additional tests generated during sketching, $AGen_{WB}$ – and the coverage criteria which drives the technique – can be further strengthened. Overall, $ASketch_{Equiv}$ combines multiple AUnit avenues together to provide another flavor of sketching Alloy models that is rooted in one of the original debugging methods for Alloy users, while leveraging the effectiveness of $ASketch_{EvalSub}$.

Chapter 8

Future Work

We take a look at three general avenues for future work based on the insights and contributions of this dissertaton.

8.1 Debugging Alloy Models

The motivation behind AUnit is to bring well established imperative testing paradigms to the declarative world, starting with Alloy. Alloy has its own tool support through the Alloy Analyzer, which allows for an automated analysis of the language. Specifically, AUnit introduces the concepts of test case, test execution and test coverage for Alloy. Efforts have been made to take the foundation of AUnit and provide: (1) the ability to automatically generate tests and (2) the ability to sketch Alloy models. However, the building blocks of AUnit facilitate other imperative testing paradigms, two notable candidates are fault localization and repair.

Fault Localization: When an AUnit test fails, one of two possibilities holds: (1) the user incorrectly specified a test or (2) the user's model is incorrect. In the first case, the Alloy Analyzer can visually display the test's valuation, enabling the user to quickly validate or correct the test. However, in the second case, the user needs to debug the Alloy model. If the model is over-constrained – rules out an instance meant to be valid – then the Alloy Analyzer does provide an initial basis for fault localization through the Analyzer's ability to highlight the unsat core. The unsat core tool-set highlights the parts of the model which together lead to an unsatisfiable solution for a particular command. When a model is over-constrained, at least one test will falsely be unsatisfiable, enabling the user to leverage the unsat core highlighting to narrow the focus for their debugging efforts. However, if the model is under-constrained – allows an invalid instance to be generated – there is currently no support for localizing the cause of the fault. The user simply knows there is a bug somewhere in the formula(s) invoked by a failing test's command. Additionally, there are limitations to the unsat core functionality. Given a formula is the form "`all d : D | a and b` " if the formula is false because "`a`" is false, the entire quantified formula will be highlighted. Therefore, regardless of the type of bug, fault localization techniques are needed.

There are two different avenues to explore for fault localization. The first is to inspect the output of the unsat core functionality to see if it is possible to get a narrower slice of the Alloy model. Second, using the valuation's coverage information, we can paint a clearer picture for the end user over what is happening when their valuation is essentially an "input" to the formulas of the test. In particular, we can explore the AUnit coverage slices of passing and failing tests, with the goal of narrowing in on the unique coverage characteristics of failing tests. Our coverage requirements look to have valuation explore different sizes of expressions and different truth values of formulas. For instance, looking at the unique coverage provided by a failing test could give insight into the misuse of transitive closure in place of reflexive transitive closure.

Repair: Given that a user has confidence in their AUnit test suite, failing test(s) can be used to *repair* a faulty model. The problem of repair centers around automatically correcting a faulty portion of code based on an error in behavior being detected. Taking inspiration from our *ASketch* framework, we can view the repair problem as a tailored and narrowly focused sketching problem. First, we can use the unsat core and fault localization strategies to isolate what parts of the Alloy model should be abstracted away into a series of sketches. These sketches would start with one hole and build up an array of different hole combinations e.g. if we want to repair `"some List.header"` we could generate 3 sketches (1) `"\UO\ List.header"`, (2) `"some \E\"`, and (3) `"\UO\ \E\"`. Additionally, we can also build on top of our $ASketch_{Gen}$ problem, directly re-using the set of candidates for all terminal holes (Table 5.1). For expression holes, `\E\`, we can generate a series of candidate expressions by mutating the given expression in the model, starting with minor changes and building up to larger mutations. Then, using these new $ASketch_{Gen}$ guidelines, we can systematically solve each sketch, stopping when one candidate solution passes all tests.

8.2 Advancements for *ASketch*$_{Eval}$

8.2.1 Exploring the Evaluator

ASketch$_{Eval}$ is able to solve sketches quickly, as a whole, because of its *ASketch*$_{EvalSub}$ optimization, in which test candidates are grouped together by their evaluation *value* over the expressions and subformulas in the sketch. When a test candidate fails a test case, we know that in combination, the values of the subformulas being sketched cause the test to fail. However, we do not know which parts of the test candidate contributed the the failure. Consider our BinaryTree formula: `all n : Node | n in BinaryTree.root.*(left + right) => n \CO\ \E\ and \UO\ \E\ and \UO\ \E\`. If this formula fails a test because of the test candidate's values for " `n \CO\ \E\`", then we would like to be able to skip all test candidates with this combination. This optimization could enable *ASketch*$_{Eval}$ to skip chunks of candidate solutions in mass, rather than taking the effort to build a candidate's representation in order to eliminate it. However, we currently have no way of knowing what part of the test candidate contributed to the failure. Future work can focus on exploring the inner workings of the `Evaluator` to isolate what specifically causes a `false` result, which can also lend itself to the fault localization problem for AUnit.

8.2.2 Automated Test Ordering

Chapter 6 highlights the importance of the test order on the time to solve the sketch. In Chapter 7, we use this information to order a test suite based on the coverage of that test suite over a user provided equivalent formula.

However, this information is not available to use in our traditional sketching format for $ASketch_{Eval}$, in which we rely on a user provided test suite to outline expected behavior rather than an equivalent formula. However, we know the importance of focusing is on high level differences between valuations in the beginning of a test order. Our AUnit valuations have a textual representation that we commonly work with; however, all valuations additionally have a graphical representation. This representation is depicted in the Alloy toolsets as a graph. Therefore, we can explore the range of graph differencing metrics and to see which metrics would be applicable to defining a process to automatically order tests based on their valuations.

8.2.3 Iterative Sketching

Moving forward, as the *ASketch* framework develops, the framework will need to adapt to an iterative viewpoint. Previously, we have mostly looked at sketching as a one off problem where a user would provide a skeleton formula with holes and a robust test suite, then the *ASketch* framework would execute and report a solution to the end user. However, in practice, sketching can be, and often is, iterative in nature. Below, we will highlight two main ways in which *ASketch* can leverage re-use in an iterative environment.

Changing the Test Suite: An end user may initially provide a test suite that fails to fully outline the intended behavior of a model-to-be-sketched. For instance, in Chapter 5, with dynamic pruning, the expression holes for the BinaryTree model have 433 possible candidates expressions each – all of which are non-equivalent and together depict a wide range of Alloy semantics and formulations. It is easy to imagine that a user may initial under describe the sketching problem, resulting in the user refining the test suite. However, rather than starting a new sketch from scratch since the test suite has changed, we can isolate what previous work we can re-use. If a user adds a test and the previously reported solution is no longer correct, we can *resume* sketching from where we initially stopped. All previously eliminated candidate solutions are not valid. However, we want to reuse the `value` mapping, which keeps track of the previous evaluation of expressions and subformulas over the tests. These evaluations are still valid and can be leveraged when looking for a new solution among the remaining candidate solutions. This approach was used in $ASketch_{Equiv}$ when a candidate solution passed all existing tests, but was not logically equivalent to the user provided equivalent predicate. If a test is removed, the reported solution is still valid, and there is no need to sketch anything. However, if a test has changed, the reported solution is not guaranteed to be correct. Assuming the reported solution is now invalid, we cannot simply resume sketching, as with the addition of a test. Instead, candidate solutions eliminated by the changed test might now be valid. Therefore; we first re-check these solutions. Once again, we can re-use the `value` store; however, we would first need to clear out the mapping for the changed test, as those evaluations may no longer hold.

Suggesting Refinements to the End User: Additionally, with Alloy, we have the ability to guide the user through refinements to their sketching test suite. In our earlier *ASketch* work in Chapter 5 and Chapter 7, our algorithms were focused on solving for the *first* solution. However, the techniques can easily be modified to look for the first x solutions or *all* solutions. With multiple solutions, we can actually use Alloy to determine if the solutions are equivalent w.r.t. each other. By definition, all reported solutions are equivalent w.r.t. the test suite used for sketching. However, the test suite might not be adequate, allowing for unintended solutions to be generated. Using Alloy, we can use a `check` command to group all potential solutions into equivalence classes. If more than one equivalence class is found, we can guide the user through eliminating the incorrect solutions by presenting the user with the counterexample that captures the difference between any two equivalence classes. In turn, the end user can label the counterexample "valid" or "invalid," producing a new test. This test can then be used to narrow down the remaining pool of solutions.

8.3 Bringing Testing to Other Declarative Language(s)

Declarative languages have commonly been used for a range of activities from a base to automatically generate imperative code to model based testing to validation of designs. In other words, traditional roles for declarative languages have often resembled providing validation for the design or implementation of other programs, often written in imperative code. However, recently, declarative languages have been used as a programming solution to problems which are defined more by a set of rules that lend themselves to declarative solutions easily, most notably through the introduction of software defined networks (SDNs). As the role of declarative languages increases, the need for better tools and support for these languages will increase. Declarative languages can take a wide range of shapes themselves, meaning that mastery of one declarative language may not give a user a head start when learning another declarative language. However, through AUnit, we have shown that it is possible to bring imperative testing paradigms to declarative languages. Furthermore, AUnit opened the door for a number of further testing advancements in Alloy from automatically generating tests [71] to mutation testing [74] to model synthesis through sketching [75]. Below we outline a user study, which can be used to derived the benefits of AUnit before moving forward to looking at other languages, as well as a guideline to the problem of SDNs, in which corporations are turning towards a split-problem approach mixing imperative and declarative languages.

User Study: The motivation behind the creation of AUnit is to ease the burden on the end-user by providing a well defined testing environment for Alloy. Furthermore, one envisioned use of the *ASketch* framework is to aid beginning Alloy users as a teaching tool. Indeed, one functionality of *ASketch* is the ability to present multiple solutions to a sketch to the user. As a result, a beginning Alloy user can be exposed to different ways in which the expressive nature of Alloy can solve the sketch. Before applying the lessons learned from building the AUnit infrastructure and *ASketch* framework to other declarative languages, we recommend a user study of both toolsets, to ensure the foundation of both techniques are effective at easing the burden of developing in Alloy. In particular, the study should focus on our notion that graphical valuations would be easy for beginning Alloy developers to reason over and create as valuations are the heart of our AUnit and *ASketch* frameworks. Then, the basis of AUnit in Alloy can be applied forward towards other declarative languages.

SDN Languages: Software defined networks, SDNs, is a new paradigm for programming in the networking field. SDNs were conceived as a "divide and conquer" mentality to the issues plaguing the networking community. Specifically, the routing protocols have changed drastically since the original implementation of the Internet. Initially, efforts were focused on an imperative implementation to handle packets. As the rules have changed, more and more imperative code has been tacked onto the initial effort, making testing difficult. With the need to re-write the packet routing code base, corporations have turned towards the use of declarative languages to handle this portion of a network, which is rule-based. Companies are willing to take on the expense of switching from an imperative base because their existing imperative code is too large and adhoc to properly test. However, as we have seen throughout this dissertation, declarative languages can be complex and hard to reason over correctly. Therefore, as different groups have begun to develop declarative-based solutions, there is a need to improve the development and test environment of such languages. Otherwise, these companies will find themselves back in the same position: unable to test their packet handling code.

Chapter 9

Conclusion

Software models play a key role in reliable systems, but writing models correctly is challenging. This dissertation addresses the problem of creating correct models of software, specifically in the context of the Alloy tool-set. We take a four-fold approach, building off of the foundations of AUnit. One, we develop automated testing techniques for Alloy, which allow users to validate their Alloy models in the traditional spirit of testing. Two, we develop a technique to synthesize parts of Alloy models based on program sketching and tests, which allow users to write partial models that are completed by automated tools. Three, we enhance the efficacy of our core sketching technique through multi-threading and exploration of test orders. Four, we leverage our automated testing techniques and our improved sketching technique to introduce a second sketching approach that requires the user to specify an equivalent formula in place of specifying a suite of test valuations. We embody our techniques in prototype tools and use them for experimental evaluation; as subjects, we use a variety of Alloy models, including some from the standard Alloy distribution and some written by graduate students.

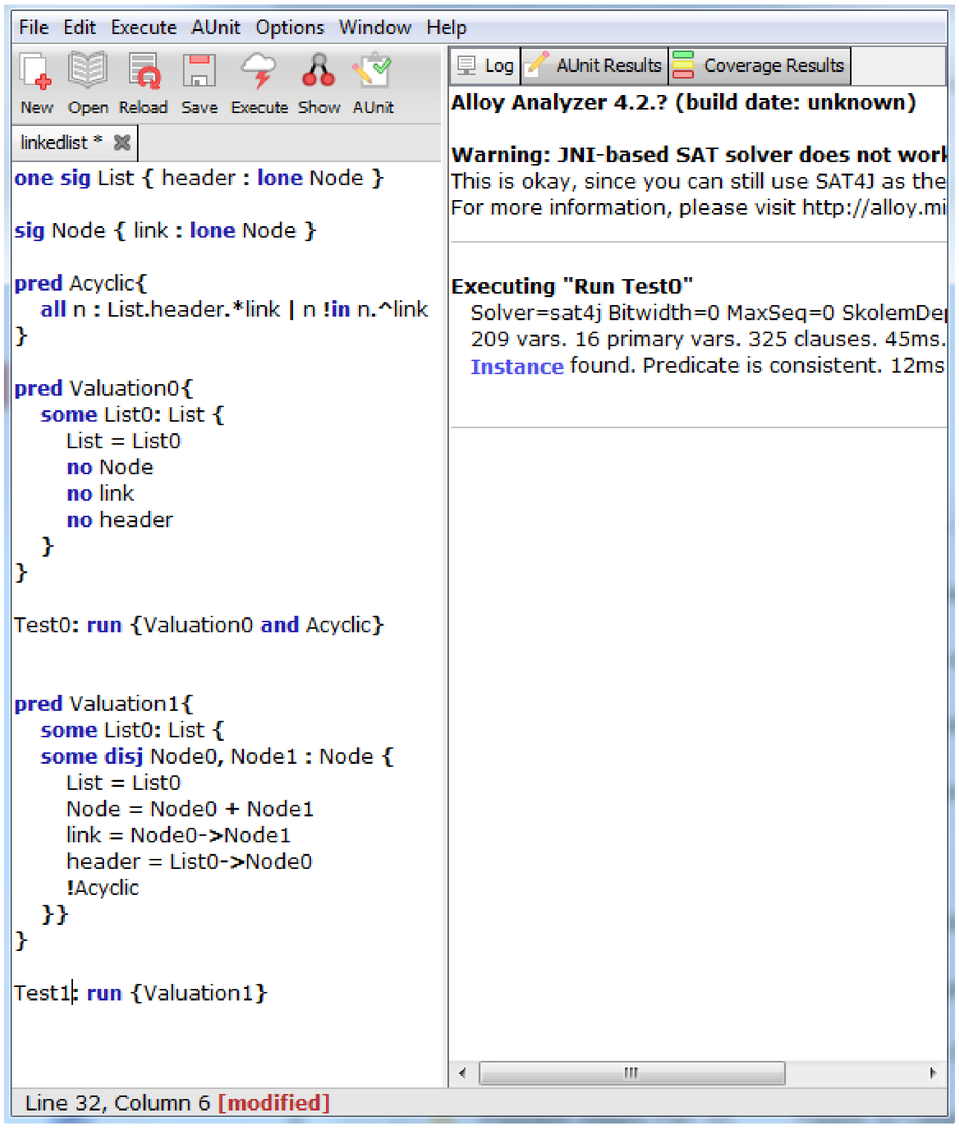
Appendices

Appendix A

Alloy Analyzer with AUnit Toolset

In this appendix section, we will highlight our embodiment of AUnit, currently deployed within the Alloy Analyzer toolset.

To start, our AUnit functionality is built on top of the Alloy Analyzer, meaning all of the original Alloy Analyzer toolset is still available. Depicted below is a screen shot from our toolset in which we have re-created our singly-linked list running example. The left hand side of the toolset is the editable portion of the tool in which the user can specify an Alloy model. The right hand side of the tool contains various reporting tabs: "`Log`" displays the current Alloy Analyzer's logging information, "`AUnit Results`" displays the test execution details for the last run AUnit test suite, and "`Coverage Results`" displays the model coverage details for the last run AUnit test suite. If we press the "`Execute`" button, the normal Alloy Analyzer functionality will take place. In this case, the labeled command "`Test0`" will be executed, as it is the first command in the model.


File Edit Execute AUnit Options Window Help
New Open Reload Save Execute Show AUnit
linkedlist *
Log AUnit Results Coverage Results

```
one sig List { header : lone Node }

sig Node { link : lone Node }

pred Acyclic{
   all n : List.header.*link | n !in n.^link
}

pred Valuation0{
   some List0: List {
      List = List0
      no Node
      no link
      no header
   }
}

Test0: run {Valuation0 and Acyclic}


pred Valuation1{
   some List0: List {
   some disj Node0, Node1 : Node {
      List = List0
      Node = Node0 + Node1
      link = Node0->Node1
      header = List0->Node0
      !Acyclic
   }}
}

Test1: run {Valuation1}
```

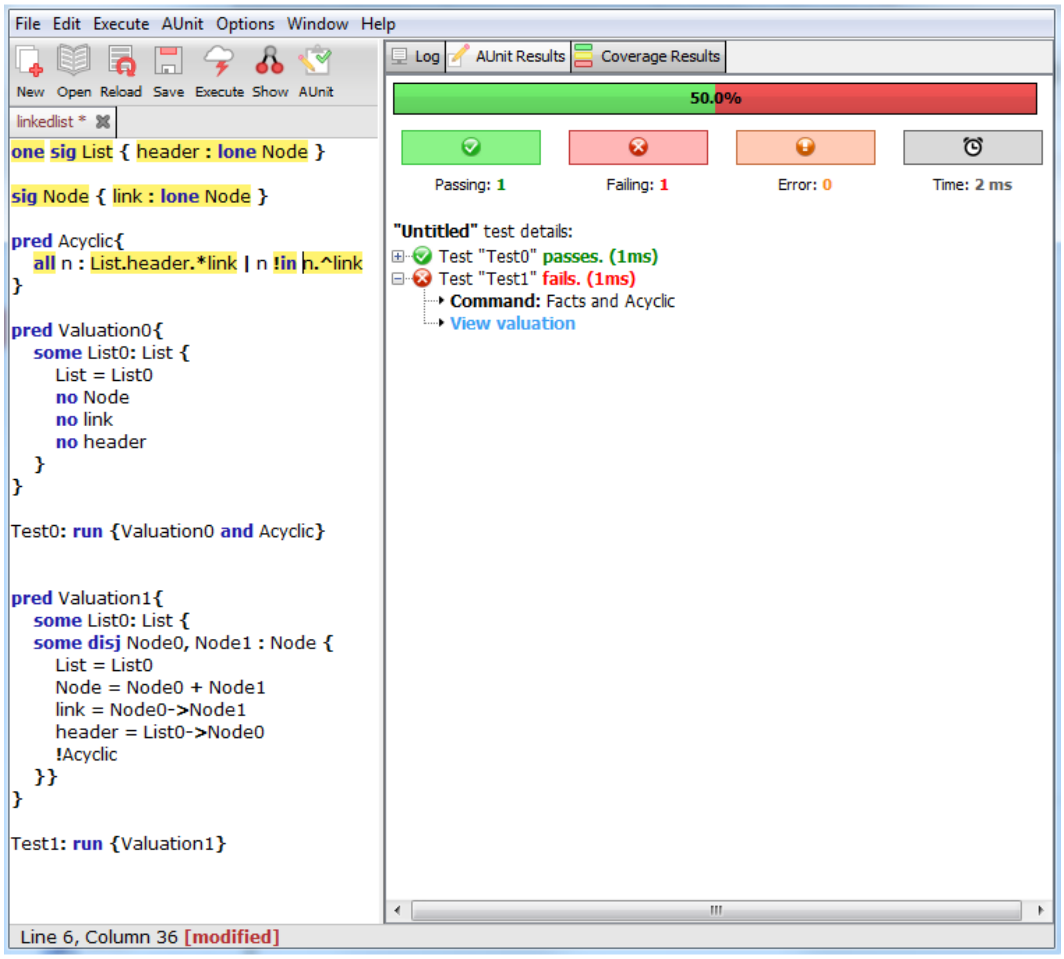
```
Alloy Analyzer 4.2.? (build date: unknown)

Warning: JNI-based SAT solver does not worl
This is okay, since you can still use SAT4J as the
For more information, please visit http://alloy.mi

Executing "Run Test0"
   Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDe
   209 vars. 16 primary vars. 325 clauses. 45ms.
   Instance found. Predicate is consistent. 12ms
```

```
Line 32, Column 6 [modified]
```

Note in this model, we have two AUnit test cases (labeled Alloy command `Test0` and `Test1`) showing the two different ways a valuation and command can be specified for the current toolset. For `Test0`, the valuation predicate (`pred Valuation0`) does not contain the command, while `Test1`'s valuation predicate (`pred Valuation1`) does contain the command. The second format is the required format to generate an AUnit test for a predicate with parameter(s), as the valuation is defined solely within its given predicate.

Next, we can execute our AUnit test suite by clicking the "`AUnit`" button. This button will execute every AUnit test present in the model, specifically by executing all commands in the model whose label contains "`Test`." The right hand side will automatically switch to the "`AUnit Results`" tab. If AUnit's coverage calculation is turned on, the left hand side of the model will be highlighted according to the coverage derived from all executed tests. Green means the Alloy construct (signature, relation, expression or formula) has been fully covered, yellow means the construct is partially covered (at least one coverage criteria meet but not all) and red means the Alloy construct has not been covered at all.

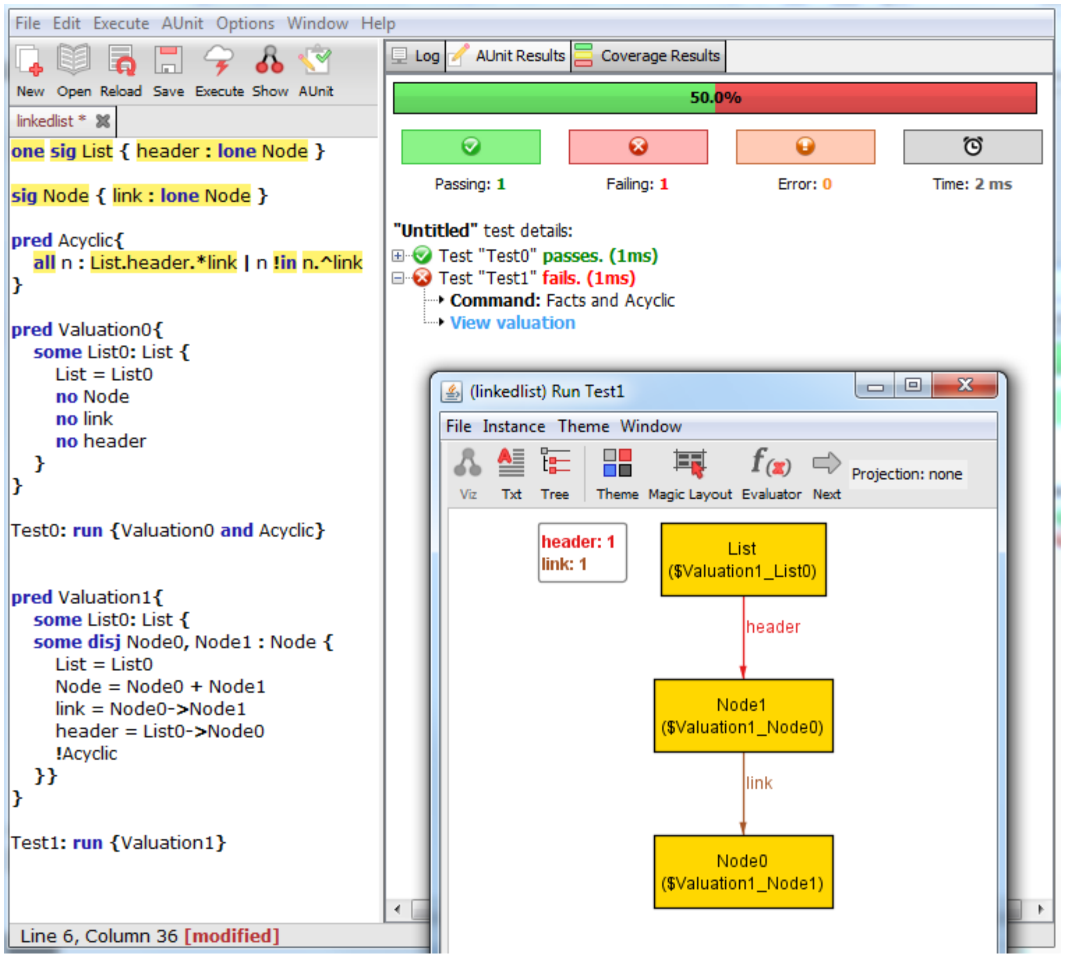
Depicted below is the behavior of executing our current AUnit test suite:


File Edit Execute AUnit Options Window Help
New Open Reload Save Execute Show AUnit
linkedlist *
Log AUnit Results Coverage Results
50.0%
Passing: 1
Failing: 1
Error: 0
Time: 2 ms
Line 6, Column 36 [modified]

```
one sig List { header : lone Node }

sig Node { link : lone Node }

pred Acyclic{
  all n : List.header.*link | n !in n.^link
}

pred Valuation0{
  some List0: List {
    List = List0
    no Node
    no link
    no header
  }
}

Test0: run {Valuation0 and Acyclic}


pred Valuation1{
  some List0: List {
  some disj Node0, Node1 : Node {
    List = List0
    Node = Node0 + Node1
    link = Node0->Node1
    header = List0->Node0
    !Acyclic
  }}
}

Test1: run {Valuation1}
```

```
"Untitled" test details:
⊞ Test "Test0" passes. (1ms)
⊟ Test "Test1" fails. (1ms)
    Command: Facts and Acyclic
    View valuation
```

Across the top, the overall summary of the test suite execution is displayed including: number of passing tests, number of failing tests, number of errors (test that ran out of memory or time), and overall time to execute the whole test suite. Then, we display the execution details associated with each test case. For a passing test, the execution time is displayed directly, but the details (the AUnit command and valuation) are initially collapsed. For a failing test, the AUnit test details start off expanded, allowing the user to immediately see what the command and valuation is. The "`View Valuation`" link graphically displays the valuation in a separate window using the Alloy Analyzer's visualization toolset, as seen below:


File Edit Execute AUnit Options Window Help
New Open Reload Save Execute Show AUnit
linkedlist *
Log AUnit Results Coverage Results
50.0%
(linkedlist) Run Test1
File Instance Theme Window
Viz Txt Tree Theme Magic Layout Evaluator Next
Projection: none
header: 1
link: 1
List ($Valuation1_List0)
header
Node1 ($Valuation1_Node0)
link
Node0 ($Valuation1_Node1)

```
one sig List { header : lone Node }

sig Node { link : lone Node }

pred Acyclic{
    all n : List.header.*link | n !in n.^link
}

pred Valuation0{
    some List0: List {
        List = List0
        no Node
        no link
        no header
    }
}

Test0: run {Valuation0 and Acyclic}


pred Valuation1{
    some List0: List {
    some disj Node0, Node1 : Node {
        List = List0
        Node = Node0 + Node1
        link = Node0->Node1
        header = List0->Node0
        !Acyclic
    }}
}

Test1: run {Valuation1}
```

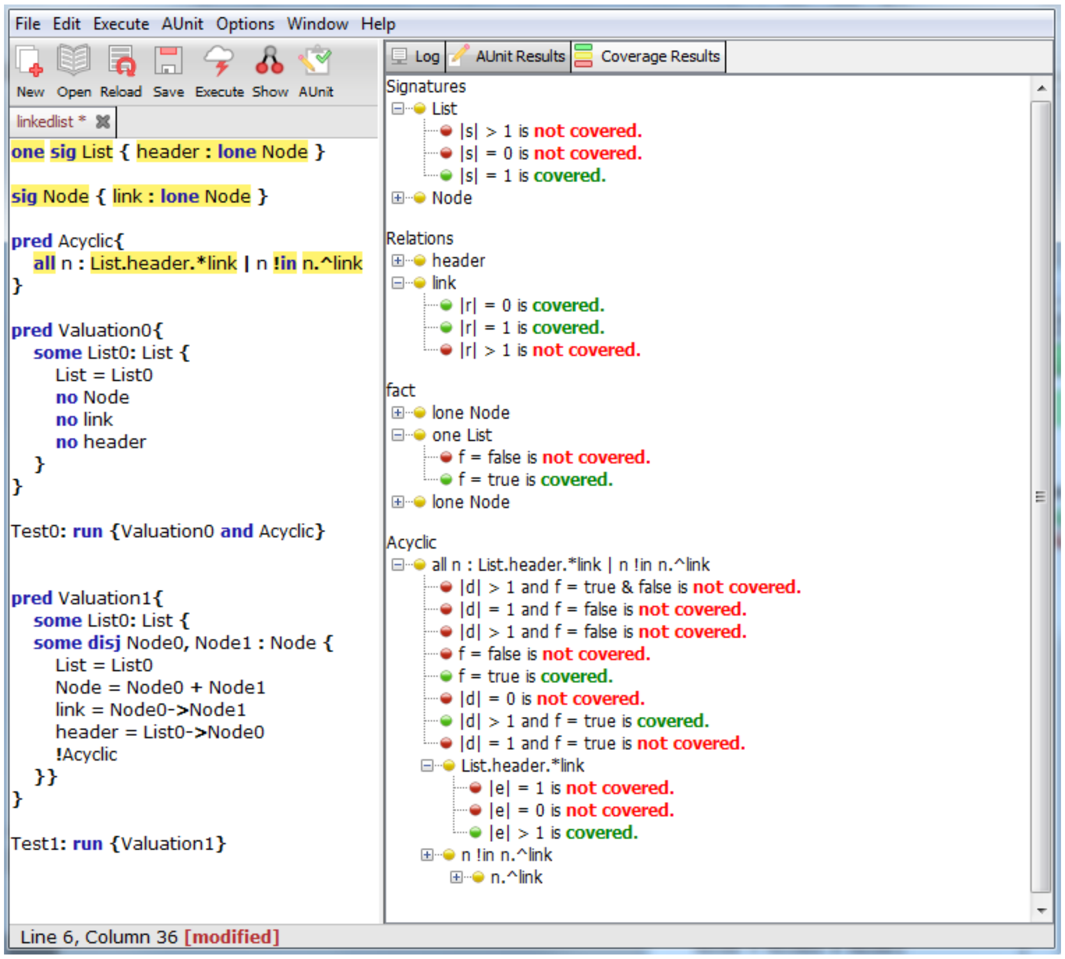
Line 6, Column 36 [modified]

Passing: 1 Failing: 1 Error: 0 Time: 2 ms

"Untitled" test details:

- Test "Test0" passes. (1ms)
- Test "Test1" fails. (1ms)
  - **Command:** Facts and Acyclic
  - View valuation

We can click over to the "`Coverage Results`" tab to view detailed coverage information. The coverage information is broken down by signatures, relations, and then predicates and assertions. For any given predicate or assertion, the tool displays all expression and formula constructs. The results start collapsed, but each Alloy construct can be expanded to display all of its coverage criteria and whether or not that criteria has been covered. Below is a screen shot showing what the expanded coverage information looks like:


File Edit Execute AUnit Options Window Help
New Open Reload Save Execute Show AUnit
linkedlist *
Log AUnit Results Coverage Results

```
one sig List { header : lone Node }

sig Node { link : lone Node }

pred Acyclic{
  all n : List.header.*link | n !in n.^link
}

pred Valuation0{
  some List0: List {
    List = List0
    no Node
    no link
    no header
  }
}

Test0: run {Valuation0 and Acyclic}


pred Valuation1{
  some List0: List {
  some disj Node0, Node1 : Node {
    List = List0
    Node = Node0 + Node1
    link = Node0->Node1
    header = List0->Node0
    !Acyclic
  }}
}

Test1: run {Valuation1}
```

```
Signatures
⊟ List
    |s| > 1 is not covered.
    |s| = 0 is not covered.
    |s| = 1 is covered.
⊞ Node

Relations
⊞ header
⊟ link
    |r| = 0 is covered.
    |r| = 1 is covered.
    |r| > 1 is not covered.

fact
⊞ lone Node
⊟ one List
    f = false is not covered.
    f = true is covered.
⊞ lone Node

Acyclic
⊟ all n : List.header.*link | n !in n.^link
    |d| > 1 and f = true & false is not covered.
    |d| = 1 and f = false is not covered.
    |d| > 1 and f = false is not covered.
    f = false is not covered.
    f = true is covered.
    |d| = 0 is not covered.
    |d| > 1 and f = true is covered.
    |d| = 1 and f = true is not covered.
  ⊟ List.header.*link
      |e| = 1 is not covered.
      |e| = 0 is not covered.
      |e| > 1 is covered.
  ⊞ n !in n.^link
    ⊞ n.^link
```

Line 6, Column 36 [modified]

Appendix B

Homework question

The following Alloy code was given as a partial Alloy model to be completed with respect to the given instructions:

```
module list

sig List {
  header: set Node
}

sig Node {
  link: set Node,
  elem: set Int
}

fact CardinalityConstraints {
   // each list has at most one header node
   /* your code goes here */

   // each node has at most one link
   /* your code goes here */

   // each node has exactly one elem
   /* your code goes here */
}

pred Loop(This: List) {
   // <This> is a valid loop-list
   /* your code goes here */
}

pred Sorted(This: List) {
   // <This> has elements in sorted order (‘<=’)
   /* your code goes here */
}

pred RepOk(This: List) {  // class invariant for List
  Loop[This]
  Sorted[This]
}
```

```
pred Count(This: List, x: Int, result: Int) {
   // count correctly returns the number of occurences of <x> in <This>
   // <result> reprsents the return value of count

  RepOk[This] // assume This is a valid list

   /* your code goes here */
}

abstract sig Boolean {}

one sig True, False extends Boolean {}

pred Contains(This: List, x: Int, result: Boolean) {
   // contains returns true if and only if <x> is in <This>
   // <result> represents the return value of contains

  RepOk[This] // assume This is a valid list

   /* your code goes here */
}
```

Appendix C

Mutation Testing for Alloy

This appendix summarizes the μAlloy approach, which we include for ease of reference; further details can be found in Wang's Masters thesis [74]. This section describes μAlloy's mutation operators, process for generating mutants, and mutation score algorithm. μAlloy generates *non-equivalent* valid Alloy models as mutants.

μAlloy operators We follow the spirit of mutation testing for imperative languages to define the mutation operators for Alloy. Specifically, μAlloy uses the production rules of the Alloy grammar [1] and defines operators to mutate different constructs, e.g., signature declarations, formulas, and commands, of the Alloy language. Table C.1 defines the mutation operators used by μAlloy. `MOR` mutates signature multiplicity (e.g. `lone sig` to `one sig`). `QOR` mutates quantifiers `all, some, no, etc.`. `UOR` and `BOR` define operator replacement for respectively unary and binary operators. For example, mutate `a.*b` to `a.^b`, and mutate `a=>b` to `a<=>b`. `IOBU` inserts an operator before an unary expression (e.g. mutate `a.b` to `a.~b`). `OD` defines operator deletion (e.g. mutate `a.* ~b` to `a.*b`). `BOE` exchanges operands for a binary operator. For example, mutate `a => b` to `b => a`. `IEOE` is similar to `BOE` with the exception that it mutates the operands of `imply-else` expression (e.g. mutate `a => b else c` to `a => c else b`). `IID` mutates integer increment and decrement.

Generating Mutants To generate mutants systematically, we iterate over all Alloy AST nodes. If a location that mutation operators can be applied to is found, then we mutate the model to create a mutant. The mutant is then checked to see if it is semantically equivalent to the original model (for the given scope). If the mutant is found to be equivalent, then the mutant is discarded. We use Alloy itself to check for equivalence, as we can write a `check` command to ask Alloy if the two models are in fact equivalent. [74] contains further details on how to structure the `check` command depending on what part of the Alloy model is mutated. Not all equivalence checking can be performed with Alloy 4.2 because the checking may require higher order solvers to solve. In such cases, we use Alloy* to check the equivalence of the original model and the mutant. Also, we set the scope of equivalence checking algorithm to 8 universally for all 7 Alloy models for higher confidence in the results.

Mutation testing algorithm Algorithm 8 gives steps to calculate the mutation score for the given AUnit test suite. First, the test suite is run against the

Table C.1: Mutation Operators

Mutation Operator	**Description**
MOR	Multiplicity Operator Replacement
QOR	Quantifier Operator Replacement
UOR	Unary Operator Replacement
BOR	Binary Operator Replacement
IOBU	Insert Operator Before Unary Expression
OD	Operator Deletion
BOE	Binary Operand Exchange
IEOE	Imply-Else Operand Exchange
IID	Integer Increment and Decrement

original model and the test execution outcome (either pass or fail) is recorded. Next, the suite is run against each mutant, and as soon as a different test execution outcome is detected, the mutant is marked as killed and the process continues to the next mutant. Finally, the mutation score for the suite is computed.

Algorithm 8: Mutation Score Algorithm

```
Input: Alloy model $m$, mutants $mus$ and AUnit test suite $ts$
Output: Mutation score
    // Run the test suite against the original model and record the
       test result. The test result shows which test case is allowed or
       disallowed in the model.
    $mTestResult \leftarrow$ runTestSuite($m$, $ts$)
    // Initialize the number of killed mutant to 0.
    $numMutantKilled \leftarrow 0$
    // Run the test suite against each mutant and see if the test
       result is different.
    foreach $mu \in mus$ do
        $muTestResult$ = runTestSuite($mu$, $ts$)
        // If the result is different, then the mutant is killed. The test
           result is considered different if a test case is allowed in the
           original model but disallowed in the mutant, or vice versa.
        if isDifferent(mTestResult, muTestResult) then
            $numMutantKilled$++
    // Compute the percentage of mutants got killed.
    return $numMutantKilled$ / $mus$.size()
```

Appendix D

Models for Evaluations

The following Alloy code shows the different Alloy models used in the evaluation section of Chapter 5 and Chapter 6.

```
module LinkedList
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
sll_1: \Q\ n : Node | n in List.header.*link => n !in n.^link
sll_2: \Q\ n : Node | n \CO\ List.header.*link => n !in n.^link
sll_3: \Q\ n : Node | n \CO\ \E\ => n !in n.^link
sll_4: \Q\ n : Node | n \CO\ \E\ => n \CO\ n.^link
sll_5: \Q\ n : Node | n \CO\ \E\ => n \CO\ \E\
}


module BinaryTree

one sig BinaryTree { root: lone Node }

sig Node { left, right: lone Node }

pred IsTree() {
--all n: Node { n in BinaryTree.root.*(left + right) => {
 -- n !in n.^(left + right) and no n.left & n.right and lone n.~(left + right) }}

bt_1: all n: Node { n in BinaryTree.root.*(left + right) => {
           n \CO\  n.^(left + right) no n.left & n.right lone n.~(left + right) }
bt_2: all n: Node { n in BinaryTree.root.*(left + right) => {
           n \CO\ \E\ no n.left & n.right lone n.~(left + right) }
bt_3: all n: Node { n in BinaryTree.root.*(left + right) => {
           n \CO\ \E\ \UO\ n.left & n.right lone n.~(left + right) }
bt_4: all n: Node { n in BinaryTree.root.*(left + right) => {
           n \CO\ \E\ \UO\ \E\ lone n.~(left + right) }
bt_5: all n: Node { n in BinaryTree.root.*(left + right) => {
           n \CO\ \E\ \UO\ \E\ \UO\ n.~(left + right) }
bt_6: all n: Node { n in BinaryTree.root.*(left + right) => {
           n \CO\ \E\ \UO\ \E\ \UO\ \E\ }
}
```

```
module Contains
one sig List { header: lone Node }

sig Node {
  elem: lone Object,
  link: lone Node
}

sig Object {}

pred Contains(l: List, e: Object) {
--e in l.header.*link.elem
contains_1: \E\ in l.header.*link.elem
contains_2: \E\ \CO\ l.header.*link.elem
contains_3: \E\ \CO\ \E\
}


module Remove
one sig List { header, header': lone Node }

sig Node {
  elem, elem': one Object,
  link, link': lone Node
}

sig Object {}

pred Remove(l: List, e: Object) {
--l.header.*link.elem - e = l.header'.*link'.elem'
remove_1: \E\ - e = l.header'.*link'.elem'
remove_2: \E\ \LO\ e = l.header'.*link'.elem'
remove_3: \E\ \LO\ \E\ = l.header'.*link'.elem'
remove_4: \E\ \LO\ \E\ = \E\
}


module Dijkstra

open util/ordering [State] as so

sig Process {}
sig Mutex {}

sig State {
  holds, waits: Process -> Mutex
}

pred Deadlock() {
  --some Process
  --some s: State | all p: Process | some p.(s.waits)
djk_1: \UO\ Process
       some s: State | all p: Process | some p.(s.waits)
djk_2: \UO\ \E\
       some s: State | all p: Process | some p.(s.waits)
djk_3: \UO\ \E\
       \Q\ s: State | all p: Process | some p.(s.waits)
djk_4: \UO\ \E\
       \Q\ s: State | \Q\ p: Process | some p.(s.waits)
djk_5: \UO\ \E\
       \Q\ s: State | \Q\ p: Process | \UO\ p.(s.waits)
djk_6:  \UO\ \E\
       \Q\ s: State | \Q\ p: Process | \UO\ \E\
}
```

Bibliography

[1] Alloy Reference. http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf.

[2] Alloy home page. `http://alloy.mit.edu`.

[3] Unified modeling language webpage. `http://www.uml.org/`.

[4] Basel Y Al-Naffouri. MintEra: A testing environment for Java programs. Masters report. Massachusetts Institute of Technology, 2004.

[5] 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 *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8, 2013.

[6] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer, 2016.

[7] Rastislav Bodík and Barbara Jobstmann. Algorithmic program synthesis: Introduction. *STTT*, 15(5-6):397–411, 2013.

[8] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

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

[11] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, pages 215–222, September 1976.

[12] François Degrave, Tom Schrijvers, and Wim Vanhoof. Logic-based program synthesis and transformation. chapter Automatic Generation of Test Inputs for Mercury, pages 71–86. Springer-Verlag, 2009.

[13] Yves Deville and Kung-Kiu Lau. Logic program synthesis. *The Journal of Logic Programming*, 19:321–350, 1994.

[14] T. T. Dinh-trong, S. Ghosh, and R. B. France. A systematic approach to generate inputs to test uml design models. In *2006 17th International Symposium on Software Reliability Engineering*, pages 95–104, Nov 2006.

[15] Trung Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. A. Andrews. A tool-supported approach to testing uml design models. In *10th IEEE*

International Conference on Engineering of Complex Computer Systems (ICECCS'05), pages 519–528, June 2005.

[16] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[17] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 599–612, 2017.

[18] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, 2015.

[19] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, New York, NY, USA, 2014. ACM.

[20] 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. *IEEE Transactions on Software Engineering*, 39(9):1283–1307, September 2013.

[21] Khalid Ghori. Constraint-based program repair. Master's thesis, Department of Electrical and Computer Engineering, University of Texas at Austin, August 2006.

[22] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 225–234, Cape Town, South Africa, 2010.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, 2005.

[24] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, pages 156–173, June 1975.

[25] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 53–62, Clearwater Beach, FL, 1998.

[26] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 418–423, 2011.

[27] Hans-Martin Horcher. Improving software tests using Z specifications. In *Proc. 9th International Conference of Z Users, The Z Formal Specification Notation*, pages 152–166, 1995.

[28] Jinru Hua and Sarfraz Khurshid. Sketch4J: Execution-driven sketching for Java. Under submission, 2017.

[29] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, 1975.

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

[31] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[32] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. JSketch: Sketching for Java. In *Proc. 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, New York, NY, USA, 2015. ACM.

[33] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proc. 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, New York, NY, USA, 2010. ACM.

[34] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 467–477, May 2002.

[35] Sarfraz Khurshid, Muhammad Zubair Malik, and Engin Uzuncaova. An automated approach for writing Alloy specifications using instances. In *Second International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, pages 449–457, 2006.

[36] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 272–286, 2003.

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

[38] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, New York, NY, USA, 2013.

[39] Bogdan Korel. Automated test data generation for programs with procedures. In *Proc. International Symposium on Software Testing and*

Analysis (ISSTA), pages 209–215, San Diego, CA, 1996.

[40] Shriram Krishnamurthi, Kathi Fisler, Daniel J. Dougherty, and Daniel Yoo. Alchemy: Transmuting base Alloy specifications into implementations. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 158–169, 2008.

[41] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. *SIGPLAN Not.*, 45(6):316–329, June 2010.

[42] N. Lazaar, A. Gotlieb, and Y. Lebbah. Fault localization in constraint programs. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 61–67, Oct 2010.

[43] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. *Principles and Practice of Constraint Programming – CP 2010: 16th International Conference, CP 2010, St. Andrews, Scotland, September 6-10, 2010. Proceedings*, chapter On Testing Constraint Programs. 2010.

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

[45] Nuno Macedo, Alcino Cunha, and Tiago Guimarães. Exploring scenario exploration. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*, pages 301–315, 2015.

[46] Muhammad Zubair Malik, Khalid Ghori, Bassem Elkarablieh, and Sarfraz Khurshid. A case for automated debugging using data structure repair. In *ASE*, 2009.

[47] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. *SIGPLAN Not.*, 40(6):48–61, June 2005.

[48] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. Conference on Automated Software Engineering (ASE)*, pages 22–31, San Diego, CA, November 2001.

[49] MIT. Alloy grammar.

[50] Vajih Montaghami and Derek Rayside. Extending alloy with partial instances. In *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, pages 122–135, 2012.

[51] Vajih Montaghami and Derek Rayside. Staged evaluation of partial instances in a relational model finder. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, pages 318–323, 2014.

[52] Tim Nelson, Salman Saghafi, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Aluminum: principled scenario exploration through

minimality. In *International Conference on Software Engineering*, pages 232–241, 2013.

[53] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proc. Second International Conference on the Unified Modeling Language*, pages 416–429, October 1999.

[54] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *SIGPLAN Not.*, 50(6):619–630, June 2015.

[55] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. *SIGPLAN Not.*, 49(6), June 2014.

[56] Orest Pilskalns, Anneliese Andrews, Andrew Knight, Sudipto Ghosh, and Robert France. Testing {UML} designs. *Information and Software Technology*, 49(8):892 – 912, 2007.

[57] C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, 1976.

[58] Jason Schimpf. Eclipse prolog unit test library. `http://eclipseclp.org/doc/bips/lib/test_util/index.html`.

[59] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT Interna-*

tional Symposium on Foundations of Software Engineering, ESEC/FSE-13, 2005.

[60] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.

[61] Ilya Shlyakhter. *Declarative Symbolic Pure Logic Model Checking*. PhD thesis, MIT, February 2005.

[62] Rishabh Singh and Sumit Gulwani. *Predicting a Correct Program in Programming by Example*, pages 398–414. Springer International Publishing, Cham, 2015.

[63] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proc. 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, New York, NY, USA, 2011.

[64] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying uml/ocl models using boolean satisfiability. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 1341–1344, March 2010.

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

[66] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. *SIGPLAN Not.*, 42(6):167–178, June 2007.

[67] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, New York, NY, USA, 2008. ACM.

[68] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[69] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[70] Allison Sullivan. AUnit – a testing framework for Alloy. Master's thesis, University of Texas at Austin, May 2014.

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

[72] Allison Sullivan, Razieh Nokhbeh Zaeem, Sarfraz Khurshid, and Darko Marinov. Towards a test automation framework for alloy. In *Proceed-*

ings of the 2014 International SPIN Symposium on Model Checking of Software, SPIN 2014, pages 113–116, 2014.

[73] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 632–647, 2007.

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

[75] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. Sketching Alloy models. Submitted for peer-review.

[76] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[77] Jason Wielemaker. Swi prolog reference manual. `http://www.swi-prolog.org/pldoc/doc_for?object=manual`.

[78] Jason Wielemaker. Swi prolog unit test framework. `http://www.swi-prolog.org/pldoc/package/plunit.html`.