The Thesis committee for Allison Sullivan
certifies that this is the approved version of the following thesis:

AUnit - A Testing Framework for Alloy

APPROVED BY

SUPERVISING COMMITTEE:

____
Sarfraz Khurshid, Supervisor

____
Dewayne Perry

AUnit - A Testing Framework for Alloy

by

Allison Sullivan, B.S.

THESIS

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

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Dedicated to my loving parents, Brian and Lori Sullivan, and my supportive best friend Traci Overstreet.

Acknowledgments

I wish to thank my supervisor Sarfraz Khurshid for his continued support and encouragement throughout the drafting of this thesis. I would also like to thank Dr. Dewayne Perry for taking the time to read and evaluate this work.

My colleague Razieh Zaeem and Prof. Darko Marinov (UIUC) provided valuable input and detailed feedback on this work. This thesis is in part based on a paper entitled "Towards a Test Automation Framework for Alloy" co-authored by myself, Razieh Nokhbeh Zaeem, Sarfraz Khurshid and Darko Marinov currently under submission for peer-review.

I am thankful for the continued support provided by the Cockrell Foundation through their bestowment of the Virginia & Ernest Cockrell, Jr. Fellowship. In addition, this work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628 and CNS-0958231).

AUnit - A Testing Framework for Alloy

Allison Sullivan, M.S.E
The University of Texas at Austin, 2014

Supervisor: Sarfraz Khurshid

Writing declarative models of software designs and analyzing them to detect defects is an effective methodology for developing more dependable software systems. However, writing such models correctly can be challenging for practitioners who may not be proficient in declarative programming, and their models themselves may be buggy. We introduce the foundations of a novel test automation framework, AUnit, which we envision for testing declarative models written in Alloy – a first-order, relational language that is supported by its SAT-based analyzer. We take inspiration from the success of the family of xUnit frameworks that are used widely in practice for test automation, albeit for imperative or object-oriented programs. The key novelty of our work is to define a basis for unit testing for Alloy, specifically, to define the concepts of *test case* and *test coverage* as well as *coverage criteria for declarative models*. We reduce the problems of declarative test execution and coverage computation to *partial evaluation* without requiring SAT solving. Our vision is to blend how developers write unit tests in commonly used programming

languages with how Alloy users formulate their models in Alloy, thereby facilitating the development and testing of Alloy models for both new Alloy users as well as experts. We illustrate our ideas using a small but complex Alloy model. While we focus on Alloy, our ideas generalize to other declarative languages (such as Z, B, ASM).

Table of Contents

List of Figures

Chapter 1

Introduction

Building software designs is a key part of software development for critical systems. Design flaws that go undetected into later stages of development can be very costly to fix. Analyzing software designs provides an effective methodology to get higher quality designs that can lead to more dependable software systems. While the last two decades have seen much progress in *analyzable* design languages [9] – à la model checking [6, 8] – the task of writing correct designs that accurately capture the key elements of the software system under development remains challenging, often requiring much manual effort on part of the practitioners. Moreover, what makes this task particularly demanding is that design languages do not always bear similarities in syntax and semantics to commonly used programming languages, and thus pose a substantial learning burden on the practitioners. Furthermore, what makes this task even harder is that tools that support writing designs often are not as advanced as those that are commonly used for writing imperative (or object-oriented) programs, and thus practitioners may employ ad-hoc and ineffective techniques in their effort to validate designs.

Our thesis is that it is feasible to facilitate automated testing of designs in the spirit of well-known and effective testing techniques that are widely used for imperative programs. Our focus is on writing software designs in the Alloy modeling language [9], which is among the first fully analyzable design languages. Alloy is a first-order declarative language based on relations. The Alloy analyzer utilizes off-the-shelf SAT technology [7] to analyze Alloy models. Given (1) an Alloy *model*, (2) a *command* in the model to execute, and (3) a *scope*, i.e., a bound on the universe of discourse, the analyzer builds a *constraint-solving problem* and uses its SAT-based backend to solve the problem.

This thesis introduces some central ideas that lay the foundation of AUnit, a novel test automation framework that we envision for testing declarative models written in Alloy. Our work takes inspiration from the success of the family of xUnit frameworks [3] that are used widely in practice for automated testing, albeit largely in the context of non-declarative programs. Our primary design goals are:

- To facilitate writing Alloy models correctly for users who are adept at commonly used programming languages but maybe new to Alloy;
- To enable more effective testing of Alloy models by providing a framework that allows adapting testing techniques that are effective in practice in the context of imperative programs.

The key novelty of our work is to define *declarative test cases* (à la unit tests for imperative code) and *model coverage* (à la code coverage for imperative code) for given test suites for Alloy models. Our key insight is that to gain confidence in the correctness of an Alloy model, it is crucial to observe some *valid* as well as some *invalid* valuations for the model. Valid valuations allow observing constraint *satisfaction*, which helps determine whether the model is *under*-constrained. In contrast, invalid valuations allow observing constraint *violation*, which helps determine whether the model is *over*-constrained. Indeed, in our personal experience of writing Alloy models over the years, we often found that bugs in our models were under-constrained or over-constrained formulas. Moreover, we routinely found ourselves validating our models by evaluating them for some given candidate valuations as well as asking Alloy to enumerate all solutions for some (very small) scope and then manually checking if the solutions were indeed all expected (i.e., no invalid valuation was generated), and if all expected solutions were generated (i.e., no valid valuation was missed).

We define a test case to be a pair $\langle \sigma, \rho \rangle$ where σ is a (partial) assignment of values to the relations in the model and ρ is an Alloy *command* that defines the constraint-solving problem. A test *passes* if σ is a (partial) solution with respect to the command ρ, and *fails* otherwise. Our definition of model coverage blends the spirit of logic-based coverage (e.g., *clause* coverage or *predicate* coverage) for imperative programs [4] with the relational nature of Alloy models where each expression is a relation, i.e., a *set* of tuples. A key novelty of our work is to introduce a number of model coverage *criteria* based on the specific structure of Alloy models as well as the specific nature of

Alloy formulas. To illustrate on a simple example, one of our criteria defines requirements for *quantified* formulas, which include requiring a *universally* quantified formula to be true (1) *vacuously* and (2) with respect to a non-empty universe.

We reduce the problems of declarative test execution and coverage computation to *partial evaluation* where Alloy formulas and expressions are evaluated for each given assignment to determine test pass/fail results and coverage requirements that are met.

We make the following contributions:

- **Unit testing for Alloy**. We introduce the idea of testing Alloy models in the spirit of unit testing of imperative code where given tests are executed to report test pass/fail and code coverage results.
- **Declarative test cases**. We formalize the definition of test cases for Alloy models and define the semantics of passing and failing of tests;
- **Model coverage**. We introduce eight criteria for computing model coverage and present a subsumption relation among the coverage criteria; and
- **Case Studies**. We demonstrate the utility of AUnit by showing how it supports some common testing scenarios in the context of writing Alloy models.

Chapter 2

Example

Figure 2.1 presents a small Alloy model of singly-linked, *acyclic* lists; specifically, the model allows multiple lists, which may share nodes, but each list individually must be acyclic. The keyword `module` names the model, which can be imported in other models.

The signature declaration `sig Node` introduces `Node` as a set of atoms and `link` as a binary relation that has the type `Node` × `Node`. The body of a signature can either be empty or introduce one or more fields, i.e. `link`. These fields serve to introduce relations between atoms. In general, a relation does not always have to be between the same types of atoms. Constraints can be placed on the possible values of any field. The relation `link` has a multiplicity constraint `set`. Therefore, a `link` relates one `Node` atom to any number of `Node` atoms.

In Alloy, a fact is always assumed to hold; therefore, facts do not need to be explicitly invoked. The *fact* (`fact`) `PartialFunction` uses *universal* quantification (`all`) to specify that each node is related to at most one node (`lone`) under the `link` relation, i.e., `link` is a partial function. In addition to universal quantification, Alloy supports existential quantification (‘`some`’) and also pro-

```
module list

sig Node { link: set Node }
fact PartialFunction { all n: Node | lone n.link }

pred NoDirectedCycles() { all n: Node | n !in n.^link }

run NoDirectedCycles // the scope is implicitly set to 3
```

Figure 2.1: Alloy model of singly-linked, *acyclic* lists.

vides short cuts to specializations of these quantifiers through the keywords (‘`no`’), (‘`one`’), and (‘`lone`’).

A predicate is a named formula, which must be invoked elsewhere in order to constrain the model. A predicate can be defined with zero or more arguments. The *predicate* (`pred`) `NoDirectedCycles` uses universal quantification to define acyclicity. The quantified formula includes a subset exclusion (`!in`) formula which will hold if the left hand argument is not a valid subset of the right hand argument. The operator ‘`^`’ is transitive closure. Conceptually, the expression `n.^link` represents the set of all nodes reachable from `n` following one or more traversals along `link`. Thus, `NoDirectedCycles` specifies that the set of nodes reachable from any node does not include that node itself.

The command `run NoDirectedCycles` instructs the analyzer to find an *instance*, i.e., a valuation of `Node` and `link` such that the fact formula and the predicate formula are true for the default scope of 3, i.e., at most 3 atoms in the set `Node`. Figure 2.2 illustrates three of the instances that are generated for this command by the analyzer. Figure 2.3 illustrates three valuations that

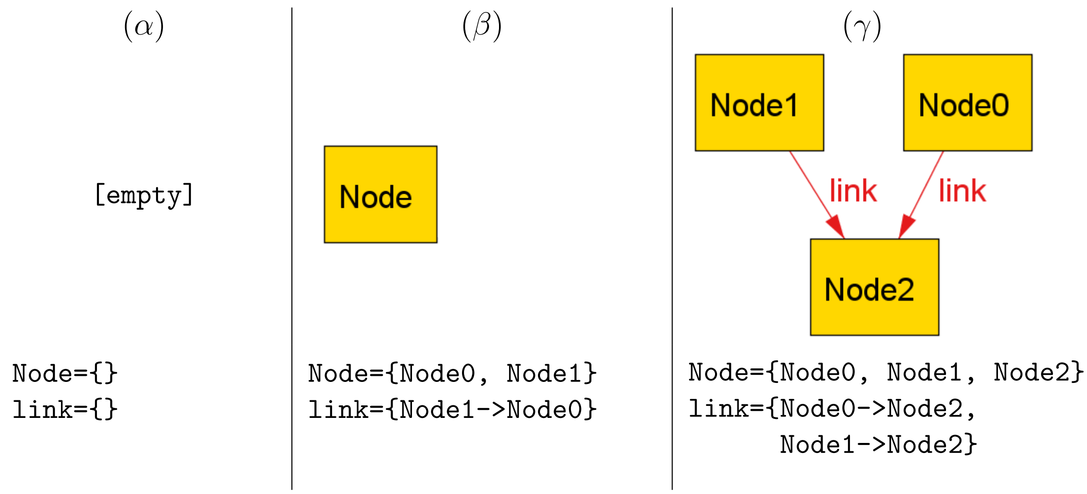

Figure 2.2: Three Alloy instances (α, β, and γ) shown graphically and textually.

are not instances and will not be generated for this command by the analyzer.

Above outlines a subset of the functionality Alloy provides. In the next chapter, we outline some additional information about Alloy. In addition, our case studies chapter introduces three more Alloy models, which use additional portions of the Alloy language. These models as well as a discussion of their key Alloy concepts can be found in Appendix A. For a full reference of the Alloy language see [1].

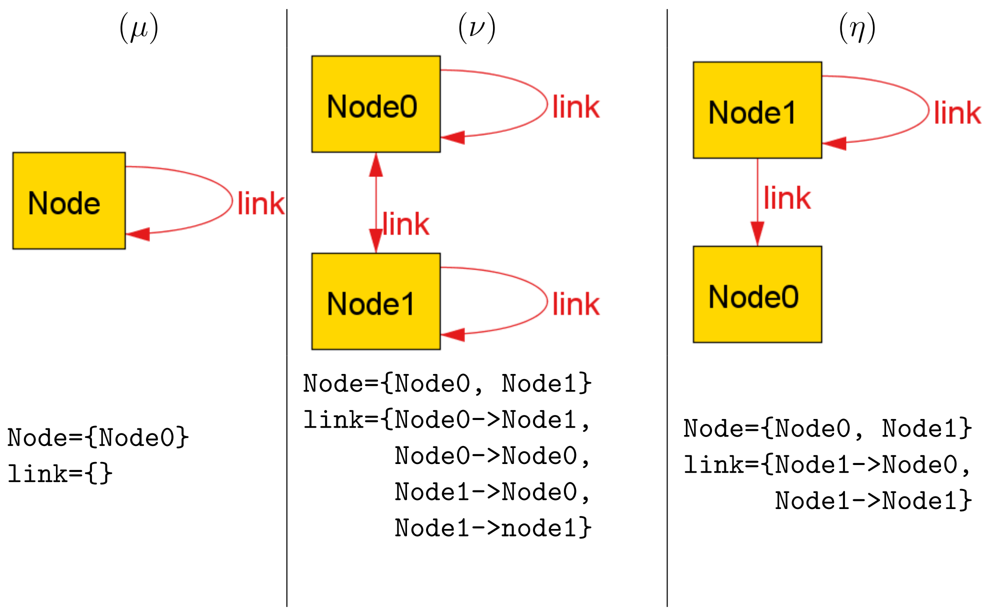

Figure 2.3: Three invalid valuations (μ, ν, and η).

Chapter 3

Background: Alloy

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

- *Signature* (`sig`). A `sig` declaration introduces a set of atoms as well as 0 or more relations.
- *Fact* (`fact`). A `fact` is a formula that must always evaluate to true for any solution generated by the Alloy Analyzer.
- *Predicate* (`pred`). A `pred` is a named (and optionally parameterized) formula, which can be *invoked* elsewhere. Alloy does not allow recursive predicates and all predicate invocations are inlined before solving.
- *Assertion* (`assert`). An `assert` is a named formula, which is intended to be *checked* for validity.
- *Command* (`run` or `check`):
  - A `run` command invokes a predicate and directs the analyzer to find an *instance*. Thus, the *constraint-solving problem* for a `run` command is to find a solution to the *conjunction* of all fact formulas and the predicate formula invoked.

– A `check` command invokes an assertion and directs the analyzer to find a *counterexample* to the assertion. Thus, the *constraint-solving problem* for a `check` command is to find a solution to the *conjunction* of all fact formulas and the *negation* of the assertion formula invoked.

Each command (implicitly or explicitly) specifies a scope, and the instances and counterexamples generated are within that scope. Moreover, each command may optionally specify an expected outcome in terms of *constraint satisfiability* using the "`expect` k" clause where $k = 0$ states the analyzer is expected to find no instance or counterexample and $k \geq 1$ states the analyzer is expected to find at least one instance or counterexample (but k does not specify the number of solutions).

Given an Alloy model and user instructions about the commands, the analyzer *executes* one or all commands in the model using Alloy's SAT-based backend and reports the constraint-solving results. If an instance or a counterexample is found, the user can inspect it in a variety of different textual and graphical formats. The user may choose to iterate through the solutions, say to enhance her/his confidence in the correctness of the model. The analyzer adds symmetry-breaking predicates to remove *isomorphic* solutions and reduce the total number of solutions [15].

Over the years, a number of extensions have been developed for Alloy. Of note are two concepts: partial instances [16] and minimal instance [14].

The initial support for partial instances came from introduction of KodKod as the SAT-based backend for the Alloy development tool, the Alloy Analyzer. A *partial instance* is provided by a user as a *partial solution* typically in the form of placing bounds on the sets within the model. Then, when a command is executed, KodKod will try to build the partial instance into a solution for the constraint solving problem [16]. Recently, an extension to the Alloy language was proposed, which is intended to ease the ability of specifying partial instances [13]. Through the introduction of partial instance paragraphs denoted by the keyword `inst`, a user can outline a partial instance using the existing Alloy grammar.

Another is the introduction of generating minimal instances through Aluminum, a modified version of Alloy. A *minimal instance* is one in which every tuple is necessary to satisfy the associated commands constraint solving problem [14]. Every instance generated by Aluminum is minimal, removing even one tuple invalidates the instance as a solution. In contrast, the Alloy Analyzer generates an instance per equivalence class defined by isomorphism breaking predicates, which can be generated automatically [15] or written manually [10, 11], and the presentation order of instances is non-deterministic. Therefore, a an instance generated by the Alloy Analyzer may or may not be a minimal instance.

Chapter 4

AUnit: Declarative Tests

A common complaint with analyzable design languages is the difficulty in producing correct specifications. Since most design languages differ in nature from popular object oriented languages (i.e., Java), programmers often encounter significant learning curves. Alloy, an expressive declarative language, is no exception. In particular when drafting an Alloy model, if a user does not explicitly forbid a behavior, then unintended features can creep into the instances or the lack thereof as a result of executing commands over the model.

Therefore, one would like to be able to have a formal methodology to check if the model behaves as expected. Currently, there is no formal basis for how a developer should check for bugs or when found, how to debug a model is missing. To fill this need, we have created a testing framework called AUnit that provides the following:

- A formal definition of a declarative test case.
- A formal definition of a test suite and what it means to execute a test suite over a given Alloy model.

- An illustration of concepts by stepping through examples.

4.1 Foundations

4.1.1 Representations of Models

We represent an Alloy model as a quintuple $\langle S, F, P, A, C\rangle$, where S is the set of all signature declarations, F is the set of all facts, P is the set of all predicates, A is the set of all assertions, and C is the set of all commands in m.

Let $m = \langle S, F, P, A, C\rangle$ be an Alloy model. Assume S is non-empty. Let Ξ be the set of all expressions other than variable declarations or uses in m. Let Φ be the set of all formulas in m. For a command $\rho \in C$, let $\Xi_\rho \subseteq \Xi$ and $\Phi_\rho \subseteq \Phi$ be the expressions and formulas respectively in the constraint-solving problem for ρ. To illustrate, for the `List` model: $\Xi = \{$`Node`, `link`, `^link`, `n.link`, `n.^link`$\}$; $\Xi_{\texttt{"run NoDirectedCycles"}} = \Xi$; $\Phi = \{$`"all n: Node | lone n.link"`, `"all n: Node | n !in n.^link"`, `"lone n.link"`, `"n !in n.^link"`$\}$; and $\Phi_{\texttt{"run NoDirectedCycles"}} = \Phi$.

Let Ξ_F, Ξ_P, and Ξ_A (each $\subseteq \Xi$) respectively be the sets of all expressions that appear in any `fact`, `predicate`, or `assertion`. Let Φ_F, Φ_P, and Φ_A (each $\subseteq \Phi$) respectively be the sets of all formulas that appear in any `fact`, `predicate`, or `assertion`.

4.1.2 Terminology

Before defining what constitutes an AUnit test case, we will first define two supporting concepts: valuation (including partial valuations) and command.

1. **Valuation:** A *valuation* is an assignment of values for the sets and relations declared in S for any given m. The valuation can either be valid or invalid for m, meaning that the assignments are not required to adhere to the constraints of the model. Specifically, a valuation is not necessarily an Alloy instance or counterexample. While the valuation will be explored as a potential solution to a constraint solving problem, the valuation exists independent from any command. On the other hand, an Alloy instance or counterexample is generated by executing a command. Therefore, an instance or counterexample is always tied to the command it is generated for.

2. **Partial Valuation:** A *partial* valuation is an assignment of values for some but not all the sets and relations declared in S for any given m. In a way, a partial valuation is actually a representation of (potentially) multiple valuations. There are a number of different ways to form partial valuations. A partial valuation can fully specify a subset of the sets and relations while leaving at least one partially specified. On the opposite end of the spectrum, a partial valuation can be written with no restrictions placed on any element in S or with some (or all) elements in S partially declared.

```
command::= [name ":"] ["run"|"check"] [name|block] scope

scope::= "for" number                          ["expect" [0|1] ]
scope::= "for" number "but" typescope,+ ["expect" [0|1] ]
scope::= "for"               typescope,+ ["expect" [0|1] ]
scope::=                                       ["expect" [0|1] ]

typescope::= ["exactly"] number [name|"int"|"seq"]
```

Figure 4.1: The Alloy Grammar of a Command, Scope, and Typescope

3. Command: In Alloy, a *command* is any `run` or `check` call that follows the syntax outlined in Figure 4.1. A `run` invokes a predicate while a `check` invokes an assertion. Alternatively, both `run` and `check` can be called with a body consisting of Alloy formulas instead of or in conjunction with an existing `pred` or `assert`. All Alloy commands have a scope. The default scope, 3, is applied to any command that does not explicitly state a scope. Executing the same `pred` or `assert` paragraph with a different scope may lead to a different outcome. In addition, a command may optionally express an expected outcome, satisfiable or unsatisfiable, using the `expect` keyword.

4.2 Declarative Test Cases

Definition 1. *A* test case *for m is a pair $\langle\sigma, \rho\rangle$ where σ is either an assignment of values to all sets and relations declared in S or a partial valuation (i.e.an assignment of values for some sets and relations declared in S but not all), and ρ is either the* empty *command (i.e., $\rho = \epsilon$) or an Alloy command that* *invokes a predicate in P or an assertion in A.*

Thus, a test case may specify just an assignment without stating any specific Alloy command. Moreover, a test case may have commands other than those that already exist in the model, i.e., belong to set C. As a direct result, a test case does not have to depend on the existence of any given `pred` or `assert`. Furthermore, the valuation need not be complete, meaning that a test case is not required to be a full description of a single potential instance. In relation to our `List` example, consider the following valuation: {`Node0`} $\subseteq$ `Node` $\subseteq$ {`Node0, Node1`}. By using the subset relation instead of the equality relation ('='), we are able to express a partial assignments of values. In this case, the set `Node` is bounded in the sense that it must have at least one `Node`, but no more than two `Node` atoms. Furthermore, as we can see, this test case expresses no constraints on the `link` relation.

Definition 2. *A* test case $t = \langle \sigma, \rho \rangle$ passes *if:*

Case 1: *σ is a complete valuation*

- *$\rho \neq \epsilon$ and σ is a solution to the constraint-solving problem for the command ρ; or*
- *$\rho = \epsilon$ and σ is a solution to the constraint-solving problem for the command "`run {} for s`" where s is the scope required for σ .*

Case 2: *σ is a partial valuation*

- $\rho \neq \epsilon$ *and there exists a solution* σ' *for command* ρ *such that* σ' *is a full assignment, which is* compatible *with the given partial assignment* σ*; or*
- $\rho = \epsilon$ *and there exists a solution* σ' *for command* `"run {} for s"` *where* s *is the scope required for* σ' *such that* σ' *is a full assignment, which is* compatible *with the given partial assignment* σ *.*

and otherwise, t fails*.*

To illustrate, let us consider a few possible test cases for our `List` model and their resulting behavior. Let σ_0 be any instance in Figure 2.2; then, the test case $\langle\sigma_0,$ `"run NoDirectedCycles"`$\rangle$ passes. On the other hand, let σ_1 be the valuation in Figure 2.3(a); then, the test case $\langle\sigma_1,$ `"run NoDirectedCycles"`$\rangle$ fails since σ_1 is not an instance of the `"run NoDirectedCycles"` command. However the test case $\langle\sigma_1,$ `"run {!NoDirectedCycles}"`$\rangle$ *passes.* The empty command, i.e. `run {}`, will accept any valuation that does not violate the `PartialFunction` fact or the signature declarations in m. Therefore, the test case $\langle\sigma_2, \epsilon\rangle$ would also pass. However, if we let σ_3 be the valuation in Figure 2.3(b), then the test case $\langle\sigma_3, \epsilon\rangle$ fails. Since `Node0` and `Node1` both have two links, the `PartialFunction` fact does not hold; therefore, σ_3 is not an instance of the empty command.

All of the above reference complete valuations. To consider the behavior of a test case using a partial valuation let σ_4 be "{`Node0`} $\subseteq$ `Node` $\subseteq$ {`Node0, Node1`}," our partial valuation discussed earlier. Consider the test case: $\langle\sigma_4,$

`"run NoDirectedCycles"`$\rangle$. The preceeding test case will pass because one enumeration of σ_4 is Figure 2.2(c) which does not contain a cycle and thus is not an instance of `NoDirectedCycles`. If we instead have a partial valuation σ_5 such that "{`Node0`} $\subseteq$ `Node` $\subseteq$ {`Node0, Node1`}" and {`Node0->Node0`} $\subseteq$ `link` $\subseteq$ {`Node1->Node1`}" then the test case $\langle\sigma_5$, `"run NoDirectedCycles"`$\rangle$ fails.

Definition 3. *A test suite is a collection of one or more test cases.*

When a test suite executes, the suite can either be successful or unsuccessful. A test suite is successful if and only if all test cases pass. Otherwise, even if only one test case fails or produces an error, the test suite is said to have run unsuccessfully.

Chapter 5

AUnit: Test Coverage

For imperative languages, developers use coverage tools such as Emma [2] for Java based programs to evaluate their current test suite. Developers can leverage coverage information to guide how they draft additional test cases or highlight test cases that are good candidates to remove from the suite.

In Alloy, now that we have a notion of a test we can gain similar advantages. A prerequisite for calculating coverage is that the Alloy model under consideration must contain an AUnit test suite. Our coverage framework serves as a good introduction to the notion of how to cover an Alloy model based on an AUnit test suite by providing the following:

- A formalization for how to calculate coverage.
- An outline of a series of requirements for covering different Alloy constructs with examples.
- An overview of eight different coverage metrics and the relationship between them.
- A comparison between coverage for declarative languages to coverage for imperative languages by focusing on Alloy and Java.

In any language, code coverage can be calculated at various levels of granularity. For instance, in Alloy, we could consider calculating the coverage of every single expression or towards the opposite end of the spectrum, we could consider calculating the coverage of every paragraph. To start, we have focused on building a coverage infrastructure primarily around two different levels: expressions and formulas. The two levels will manifest into eight different coverage metrics.

There are two possible outcomes for a test case: passing and failing. In both cases, we are able to obtain coverage information. A test case's valuation has values assigned to some if not all of the sets and relations in the Alloy model. As a result, these valuations are the tangible aspect that enables us to measure coverage. Therefore, all test cases are within the scope of our coverage framework.

5.1 Coverage computation

Let T be a test suite.

5.1.1 Coverage: test case

Let $t = \langle \sigma, \rho \rangle \in T$ be a test case.

Definition 4. *Assume $\rho \neq \epsilon$. The* coverage obtained *for t is a pair of maps $\langle \pi_t, \omega_t \rangle$ where:*

- *π_t maps each Alloy expression in Ξ_ρ to the set(s) of tuples it evaluates* *to for assignment σ; and*

- ω_t *maps each Alloy formula in* Φ_ρ *to the boolean value(s) it evaluates to for assignment* σ.

To illustrate, let σ be the instance shown in Figure 2.2(b) and $\rho =$ `"run NoDirectedCycles"`. Then $\pi_{\langle\sigma,\rho\rangle}$ is:

```
Node={Node0, Node1},
link={Node1->Node0},
^link={Node1->Node0},
n.link={{}, {Node0}},
n.^link={{}, {Node0}}
```

To clarify, the expression `n.link` is mapped to `{{}, {Node0}}` since `Node0.link={}` and `Node1.link={Node0}`.

Moreover, $\omega_{\langle\sigma,\rho\rangle}$ is:

```
"all n: Node | lone n.link"=true,
"all n: Node | n !in n.^link"=true,
"lone n.link"=true
"n !in n.^link"=true
```

To clarify, the formula `"lone n.link"` is mapped to `true` since `"lone Node0.link" = true` and `"lone Node1.link" = true`.

Additionally, let us consider the test case where σ is instance shown in Figure 2.2(c) and $\rho =$ `"run {!NoDirectedCycles"}`. Then $\pi_{\langle\sigma,\rho\rangle}$ is:

```
Node={Node0, Node1},
link={Node1->Node0, Node1->Node1},
^link={Node1->Node0, Node1->Node1},
n.link={{}, {Node0, Node1}},
n.^link={{}, {Node0, Node1}}
```

Once again, the expression `n.link` is mapped to {{}, {`Node0, Node1`}} since `Node0.link=`{} and `Node1.link=`{`Node0, Node1`}. Similarly, expression `n.^link` is also captured as set of sets. Expressions which deal with variables (i.e. `n`) need to account for all possible inputs. In this case, `n` is populated with all the set `Node`.

Furthermore, $\omega_{\langle\sigma,\rho\rangle}$ is:

```
"all n: Node | lone n.link"=true, false
"all n: Node | n !in n.^link"=true, false
"lone n.link"=true, false
"n !in n.^link"=true, false
```

In the previous example, all formulas simply evaluated to `true`; however, that is not the case here. For this test case, all four formulas evaluate to both true and false. Consider the formula `"lone n.link"`, the formula is mapped to `true, false` since `"lone Node0.link" = true` but `"lone Node1.link" = false`.

Definition 5. *Assume* $\rho = \epsilon$. *Let command* $c \in C$. *Let the coverage obtained for test* $\langle\sigma, c\rangle$ *be* $\langle\pi_{t_c}, \omega_{t_c}\rangle$. *Then, the* coverage obtained *for* t *is a pair of maps* $\langle\pi_t = \cup_{c\in C}\pi_{t_c}, \omega_t = \cup_{c\in C}\omega_{t_c}\rangle$.

The above definition holds whether σ is a complete or partial valuation. However, it should be noted that when a test case involves a partial valuation, we will need to calculate coverage for each enumerated instance. In other words, π_t and ω_t represents a mapping for potentially multiple different instances instead of just the static information gained from one complete valuation.

5.1.2 Coverage: test suite

Often times, we are not solely concerned about the coverage provided by a single test case but the coverage provided by a test suite as a whole.

Definition 6. *The* coverage obtained *for test suite T is a pair of maps* $\langle \pi_T = \cup_{t \in T} \pi_t, \omega_T = \cup_{t \in T} \omega_t \rangle$.

5.2 Coverage Criteria

The basis of our model coverage criteria are four sets of coverage *requirements* – three (R_0, R_1, and R_2) based on Alloy *expressions* and one (R_3) based on Alloy *formulas*. To illustrate all four requirements, we will construct a test suite using the 6 valuations found in both Figure 2.2 and Figure 2.3.

5.2.1 R0: Signatures

R_0 – For each signature declaration in S, there are three requirements on the basic set s in the signature declaration:

1. $|s| = 0$;
2. $|s| = 1$; and
3. $|s| \geq 2$.

The R_0 requirements meet w.r.t. the only signature, `Node`, are as follows:

Test Case	set Node	Coverage
$\langle\alpha, \epsilon\rangle$	`Node={}`	$\lvert s\rvert = 0$
$\langle\beta, \epsilon\rangle$	`Node={Node0}`	$\lvert s\rvert = 1$
$\langle\gamma, \epsilon\rangle$	`Node={Node0, Node1, Node2}`	$\lvert s\rvert \geq 2$
$\langle\mu, \epsilon\rangle$	`Node={Node}`	$\lvert s\rvert = 1$
$\langle\nu, \epsilon\rangle$	`Node={Node0, Node1}`	$\lvert s\rvert \geq 2$
$\langle\eta, \epsilon\rangle$	`Node={Node0, Node1}`	$\lvert s\rvert \geq 2$

For the `list` example, R_0 has a total of 3 requirements seeing as there is only one set `Node`. Looking at the above test cases, in order to create a test suite that provides full coverage in relation to R_0, the suite needs to include $\langle\alpha, \epsilon\rangle$. Therefore, one possible test suite to cover R_0 would be $\{\langle\alpha, \epsilon\rangle$, $\langle\gamma, \epsilon\rangle$, $\langle\mu, \epsilon\rangle\}$.

5.2.2 R1: Relations

R_1 – For each signature declaration in S, for each relation r (i.e., non-basic set) declared in S, there are three requirements on r:

1. $|r| = 0$;
2. $|r| = 1$; and
3. $|r| \geq 2$.

To see if a test suite can be created that will satisfy all R_1 requirements for `List`, we will consider the values of each test case's `link` relation:

Test Case	Relation	Coverage
$\langle\alpha, \epsilon\rangle$	`link={}`	$\lvert r\rvert = 0$
$\langle\beta, \epsilon\rangle$	`link={}`	$\lvert r\rvert = 0$
$\langle\gamma, \epsilon\rangle$	`link={Node0->Node2, Node1->Node2}`	$\lvert r\rvert \geq 2$

$\langle \mu, \epsilon \rangle$	`link={Node->Node}`	$\lvert r \rvert = 1$
$\langle \nu, \epsilon \rangle$	`link={Node0->Node0, Node0->Node1, Node1->Node0, Node1->Node1}`	$\lvert r \rvert \geq 2$
$\langle \eta, \epsilon \rangle$	`link={Node1->Node0, Node1->Node1}`	$\lvert r \rvert \geq 2$

For the `List` example, R_1 has a total of 3 requirements, since `List` only has one relation: `link`. Requirement #2 is only satisfied by test case $\langle \mu, \epsilon \rangle$;therefore, we need this test case within our suite. To satisfy all requirements, one possible test suite is $\{\langle \alpha, \epsilon \rangle, \langle \mu, \epsilon \rangle, \langle \nu, \epsilon \rangle\}$. In addition, our previous test suite for R_0 would also handle all three requirements.

5.2.3 R2: Expressions

R_2 – For each expression $e \in \Xi_F \cup \Xi_P \cup \Xi_A$, there are three requirements on e:

1. $|e| = 0$;
2. $|e| = 1$; and
3. $|e| \geq 2$.

For the `list` example, R_2 has a total of 15 requirements – three each for the five expressions `Node`, `link`, `^link`, `n.link`, and `n.^link`. Note the 3 requirements on `link` in R_2 are the same as R_1; however, if the relation `link` was not an expression in the fact `PartialFunction` or the predicate `NoDirectedCycles`, this overlap in R_1 and R_2 would not exist. In addition, we can see that `^link` is an expression in addition to `n.^link`. Nested expressions as well as nested formulas will be considered separately.

1. Test Case: $\langle \alpha, \epsilon \rangle$

Expression	Coverage
`Node={}`	$\lvert e \rvert = 0$
`link={}`	$\lvert e \rvert = 0$
`^link={}`	$\lvert e \rvert = 0$
`n.link={}`	$\lvert e \rvert = 0$
`n.^link={}`	$\lvert e \rvert = 0$

2. Test Case: $\langle \beta, \epsilon \rangle$

Expression	Coverage
`Node={Node0}`	$\lvert e \rvert = 1$
`link={}`	$\lvert e \rvert = 0$
`^link={}`	$\lvert e \rvert = 0$
`n.link={}`	$\lvert e \rvert = 0$
`n.^link={}`	$\lvert e \rvert = 0$

3. Test Case: $\langle \gamma, \epsilon \rangle$

Expression	Coverage
`Node={Node0, Node1, Node2}`	$\lvert e \rvert \geq 2$
`link={Node0->Node2, Node1->Node2}`	$\lvert e \rvert \geq 2$
`^link={Node0->Node2, Node1->Node2}`	$\lvert e \rvert \geq 2$
`n.link={{Node2}, {Node2}, {}}`	$\lvert e \rvert = 0$, $\lvert e \rvert = 1$
`n.^link={{Node2}, {Node2}, {}}`	$\lvert e \rvert = 0$, $\lvert e \rvert = 1$

4. Test Case: $\langle \mu, \epsilon \rangle$

Expression	Coverage
`Node={Node0}`	$\lvert e \rvert = 1$
`link={Node0->Node0}`	$\lvert e \rvert = 1$
`^link={Node0->Node0}`	$\lvert e \rvert = 1$
`n.link={Node0}`	$\lvert e \rvert = 1$
`n.^link={Node0}`	$\lvert e \rvert = 1$

5. Test Case: $\langle \nu, \epsilon \rangle$

Expression	Coverage
`Node={Node0, Node1}`	$\lvert e \rvert \geq 2$
`link={Node0->Node0, Node0->Node1, Node1->Node0, Node1->Node1}`	$\lvert e \rvert \geq 2$
`^link={Node0->Node0, Node0->Node1, Node1->Node0, Node1->Node1}`	$\lvert e \rvert \geq 2$
`n.link={{Node0, Node1}, {Node0, Node1}}`	$\lvert e \rvert \geq 2$
`n.^link={{Node0, Node1}, {Node0, Node1}}`	$\lvert e \rvert \geq 2$

6. Test Case: $\langle \eta, \epsilon \rangle$

Expression	Coverage
`Node={Node0, Node1}`	$\lvert e \rvert \geq 2$
`link={Node1->Node0, Node1->Node1}`	$\lvert e \rvert \geq 2$
`^link={Node1->Node0, Node1->Node1}`	$\lvert e \rvert \geq 2$
`n.link={{}, {Node0, Node1}}`	$\lvert e \rvert = 0$, $\lvert e \rvert \geq 2$
`n.^link={{}, {Node0, Node1}}`	$\lvert e \rvert = 0$, $\lvert e \rvert \geq 2$

Given the above test cases and their associated coverage, we can construct a number of different test suites which can satisfy R_2 in total. To meet R_2, we need a test suite that covers all expressions and not just one. Therefore, one minimal test suite is $\{\langle \alpha, \epsilon \rangle, \langle \mu, \epsilon \rangle, \langle \eta, \epsilon \rangle\}$.

5.2.4 R3: Formulas

R_3 – For each formula $f \in \Phi_F \cup \Phi_P \cup \Phi_A$, there are two requirements on f:

1. f is true; and
2. f is false.

Moreover, if f is a *quantified* formula, say "$Q\ x:\ d \mid b$" with quantifier Q, variable x, domain d, and body b, there are six *additional* requirements on f:

1. $|d| = 0$;
2. $|d| = 1$ and b is true;
3. $|d| = 1$ and b is false;
4. $|d| \geq 2$ and b is true for each atom in d;
5. $|d| \geq 2$ and b is false for each atom in d; and
6. $|d| \geq 2$, b is true for at least one atom in d, and b is false for at least one atom in d.

While requirement #1 for quantified formulas (i.e., $\#d = 0$) seems to be redundant in the presence of requirement #1 for R_2, R_3 may be applied independently of R_2, and hence we have six requirements for quantified formulas.

For the `List` example, R_3 has a total of 20 requirements – two each for the four formulas `"all n: Node | lone n.link"`, `"all n: Node | n !in n.^link"`, `"lone n.link"`, `"n !in n.^link"`, and additionally six each for the two quantified formulas. To draft a test suite to meet R_3 for all formulas, we first need to see how each test case's valuation impacts the result of the formulas. The quantified formulas will list at least two values: the over `true` or `false` value and the additional requirement meet.

1. Test Case: $\langle \alpha, \epsilon \rangle$

Formula	Coverage
`all n : Node | lone n.link`	b = true, $|d| = 0$
`all n : Node | n !in n.^link`	b = true, $|d| = 0$
`lone n.link`	f = true
`n !in n.^link`	f = true

2. Test Case: $\langle \beta, \epsilon \rangle$

Formula	Coverage
`all n : Node | lone n.link`	b = true, $|d| = 1$
`all n : Node | n !in n.^link`	b = true, $|d| = 1$
`lone n.link`	f = true
`n !in n.^link`	f = true

3. Test Case: $\langle \gamma, \epsilon \rangle$

Formula	Coverage
`all n : Node | lone n.link`	b = true, $|d| \geq 2$
`all n : Node | n !in n.^link`	b = true, $|d| \geq 2$
`lone n.link`	f = true
`n !in n.^link`	f = true

4. Test Case: $\langle \mu, \epsilon \rangle$

Formula	Coverage
`all n : Node | lone n.link`	b = true, $|d| = 1$
`all n : Node | n !in n.^link`	b = false, $|d| = 1$
`lone n.link`	f = true
`n !in n.^link`	f = false

5. Test Case: $\langle \nu, \epsilon \rangle$

Formula	Coverage

`all n : Node	lone n.link`	b = false, $|d| \geq 2$
`all n : Node	n !in n.^link`	b = false, $|d| \geq 2$
`lone n.link`	f = false
`n !in n.^link`	f = false

6. Test Case: $\langle \eta, \epsilon \rangle$

Formula	Coverage
`all n : Node	lone n.link`	b = true, false, $|d| \geq 2$
`all n : Node	n !in n.^link`	b = true, false, $|d| \geq 2$
`lone n.link`	true, f = false
`n !in n.^link`	true, f = false

When we try to construct a test suite to meet all 20 requirements, we run into an issue. For the formula "`all n : Node | lone n.link`" and the criteria "$|d| = 1$ and b is true", no current test case covers this situation. As it turns out, this ends up being an infeasible requirement. In order for "$|d| = 1$ and b is true" to hold for the given formula, we would need an Alloy valuation with the following: `Node={Node0}` and `link={Node0->Node0, Node0->Node0}`. However, Alloy does not allow multiple edges that are identical. Therefore, there is no way to have two of the exact same link relation count as two links instead of one. As a result, it is impossible to satisfy the desired requirement. Yet, we can still draft a test suite that covers all requirements except the additional requirement #3 for formula #1. The following test suite covers all feasible requirements of R_3: $\{\langle \alpha, \epsilon \rangle$, $\langle \beta, \epsilon \rangle$, $\langle \gamma, \epsilon \rangle\}$, $\langle \mu, \epsilon \rangle\}$, $\langle \nu, \epsilon \rangle\}$, $\langle \eta, \epsilon \rangle\}$. For the first time, our test suite involves all 6 of our test cases.

5.2.5 Infeasible Criteria

When trying to make a test suite which satisfied R_3 for our `List` model, we discovered that there was a criteria which could not be meet by any test case. This was a result of a universal quantification formula and the `lone` multiplicity constraint being invoked is such a way that it was impossible to satisfy all formula requirements.

Infeasible criteria can come up in a number of ways outside of `all` being arranged with `lone`. For instance, had the formula been "`all n : Node | set n.link`" then we would have run into another infeasible situation. The multiplicity `set` refers to "any number," which includes zero. Therefore, there is no way to violate "`set n.link`". As a result, we would run into a number of infeasible criteria from "`set n.link`" never evaluating to false to "`all n : Node | set n.link`" also not evaluating to false, which includes all the additional requirements in which 'b' has to be false for any domain element. However, had the formula been "`all n : Node | one n.link`" there would be no infeasible requirements.

Infeasible requirements are not restricted to formulas they can occur in relation to all Alloy coverage constructs: signatures, relations, expressions or formulas. Ideally, since no test case can ever meet an infeasible requirement, there is no need to count the criterion towards coverage calculations. Counting the requirement essentially make the failure to satisfy the criteria reflect negatively even though there is no way to meet the requirement. Unfortunately, there are a number of different ways infeasible requirements can come about. Prior to actively exploring and calculating coverage, it may not be known which requirements are infeasible. When it comes to tool support for coverage, bridging the gap from knowing infeasible requirements exist to knowing which requirements are infeasible will be needed.

5.3 Coverage Metrics

Based on the four requirements outlined above, we derive the following eight coverage metrics:

Coverage Metric 1. Signature coverage (SC)*:* R_0

Coverage Metric 2. Relation coverage (RC)*:* $R_0 \cup R_1$

Coverage Metric 3. Expression coverage (EC)*:* $R_0 \cup R_1 \cup R_2$

Coverage Metric 4. Fact coverage (FaC)*:* R_3 *restricted to formulas* Φ_F*.*

Coverage Metric 5. Predicate coverage (PC)*:* R_3 *restricted to formulas* Φ_P*.*

Coverage Metric 6. Assert coverage (AC)*:* R_3 *restricted to formulas* Φ_A*.*

Coverage Metric 7. Formula coverage (FC)*:* R_3

Coverage Metric 8. Model coverage (MC)*:* $EC \cup FC$

The question now arises, is it possible to generate a test suite such that each metric is fully covered? We will attempt to generate such a test suite using the test cases outlined when going over coverage requirements for our running example. Earlier, we detailed how each test case serves to meet the requirements (R_0, R_1, R_2, R_3). Since every coverage metric is based on one or more of these requirements, we can re-use this work to derive a quality test suite. To start, we can consider our test suite we generated for R_0, which will provide full signature coverage: $\langle \alpha, \epsilon \rangle \quad \langle \gamma, \epsilon \rangle \quad \langle \mu, \epsilon \rangle$

When we look into relation coverage, we are now focused on satisfying R_1. Fortunately, our current test suite also meets all the requirements for R_1 and thus provides full relation coverage. However, our current test suite is missing two feasible requirements for R_3. Therefore we can add $\langle \nu, \epsilon \rangle$ to our test suite and now have full expression coverage. Next, we have a series of coverage metrics based on formulas. As it turns out, to fully cover all formulas (FC), our test suite needs to contain all 6 test cases. As mentioned earlier, there is one infeasible criteria for fact coverage from `List`'s `PartialFunction` which transfers over into the formula coverage. We end up with the following test suite:

$$\begin{array}{ccc} \langle \alpha, \epsilon \rangle & \langle \beta, \epsilon \rangle & \langle \gamma, \epsilon \rangle \\ \langle \mu, \epsilon \rangle & \langle \nu, \epsilon \rangle & \langle \eta, \epsilon \rangle \end{array}$$

Which will provide the coverage outlined in Figure 5.1.

5.4 Relationship Between Coverage Metrics

Three of our coverage metrics are strictly based on expressions: SC, RC, and EC. On the other hand, four are strictly based on formulas: AC,

Coverage Metric	# Req. Covered	# of Feasible Req.	Coverage
Signature Coverage	3	3	100%
Relation Coverage	3	3	100%
Expression Coverage	15	15	100%
Fact Coverage	9	10-1	100%
Predicate Coverage	10	10	100%
Formula Coverage	19	20-1	100%

Figure 5.1: Coverage for Extended Test Suite

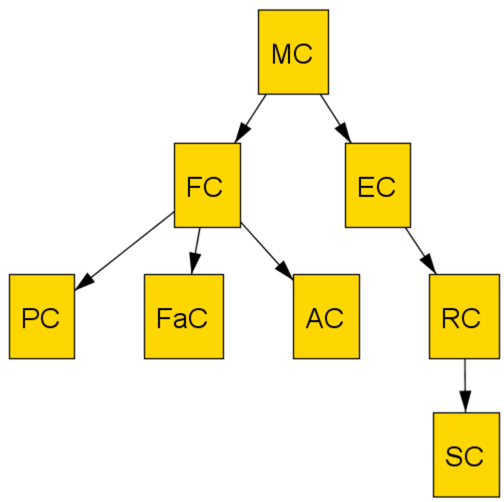

Figure 5.2: Coverage criteria subsumption relation.

FaC, FC, and PC. From their definitions, we can see how these metrics relate to one another. The coverage metrics based on two different Alloy components come together for model coverage, MC, which encompasses both EC and FC. Therefore, our eight coverage criteria satisfy the following *subsumption* partial-order '$\preceq$':

- $SC \preceq RC \preceq EC \preceq MC$

- $FC \preceq MC$,
- $FaC \preceq FC$,
- $PC \preceq FC$, and
- $AC \preceq FC$

Figure 5.2 illustrates the subsumption relation, which was generated using the Alloy analyzer; Appendix A.4 gives the corresponding Alloy model.

Earlier when outlining R_3, we mentioned how the first requirement, $|d| = 0$ at first appears redundant given that d is an expression and R_2 contains the criteria $|e| = 0$. However, we noted that R_3 can be applied independently from R_2. This is further supported by our subsumption relationship in which there is no connection between EC, which is over R_2, and FC, which is over R_3. Therefore, a test suite which completely satisfies R_3 is not guaranteed to satisfy R_2 and vice versa. As a result, the seemingly "redundant" criteria is actually required.

5.5 Comparing Code Coverage for Java and Alloy

Now that we have created our coverage framework for Alloy, we can compare and contrast our process with the well-known process of calculating coverage for imperative languages. Specifically, we will focus on comparing and contrasting the coverage process for Java, an imperative language, with Alloy, a declarative language.

Below we list several of the key similarities between covering code in a Java setting and covering code in an Alloy setting:

- Coverage is still an overall viewpoint provided by a test suite. The coverage of test cases gets summed up according to the rules of the coverage metric.
- In Java, it can be tricky to measure the coverage of a loop. As a result, people have developed commonly used guidelines for handling loops, in particular large and infinite loops. Our methodology for handling the coverage of quantifier formulas in Alloy is similar in nature to techniques used for covering loops i.e., skipping a loop, iterating over its body exactly once, and iterating over its body more than once.
- In Java, the coverage metrics fit together in the sense that some metrics subsume the other (i.e. path coverage subsumes branch coverage). For our Alloy metrics, that same relationship between different coverage metrics applies, showing that similar to Java code coverage metrics our Alloy model coverage metrics grow into more robust versions.

Below lists several of the key differences between covering code in a Java setting and covering code in an Alloy setting:

- There are a range of different common code coverage metrics for imperative code. One is statement coverage, which considers whether or not the entire statement of a program has been covered. Alloy specifications are very rich and each line can be extremely expressive. As a result, each line of an Alloy model is further broken down into formulas and/or expressions for our coverage metrics.

- In Alloy, for a single test case in which a partial valuation is involved there might exist multiple instances each of which provides coverage information. In an imperative language such as Java, a test case leads to one defined execution path, assuming the program under test is sequential.

Chapter 6

Case Studies

In this section, we will explore a number of common usage scenarios for both our testing framework as well as our coverage framework. Below, two testing scenarios and one coverage scenario are explored.

6.1 Scenario 1: Discovering a Bug in an Alloy Model

When a test case fails, one reason can be that there is a flaw in the Alloy model under test. To depict the methodology of debugging an Alloy model, we will consider the `farmers` model shown in Figure A.1. The model captures a common logic problem in which a person object (the farmer) has to get all 3 remaining types of objects (fox, chicken, and grain) to the other side of the river without one object eating another. When the farmer is present on any given side of the river, no eating occurs.

The `farmers` model is one of the models distributed in the Alloy Analyzer. For this scenario, we take a faulty version of the model, which was part of the Alloy distribution originally, but was later discovered (not by us) to have a bug, which was subsequently fixed and is a part of the current Alloy distribution. We use the faulty version and the fixed version to illustrate how writing tests could help in testing this model.

To start, we need to draft a test suite. Our first step is to determine which valuations we wish to consider. Figure 6.2 is a valuation intended to capture a solution to the `farmers` logic problem. The next four valuations are all invalid. Each valuation makes a faulty move that different portions of the model should be able to prevent. To organize our test suite, we will extend the `farmers` model to contain a series of `pred` and `assert` paragraphs that range from solving the logic problem to ensure violations do not occur. These paragraphs will feed into the commands of our test suite. With these valuations and paragraphs in place, we can now draft the following test suite:

Test1: ⟨Figure 6.2, "`run solvePuzzle for 8 State`"⟩
Test2: ⟨Figure 6.3, "`check cantAbandonAll`"⟩
Test3: ⟨Figure 6.4, "`check noQuantumObjects`"⟩
Test4: ⟨Figure 6.5, "`check farmerCantTakeFoxFirst`"⟩
Test5: ⟨Figure 6.6, "`check farmerTakesAtMostOne`"⟩

When we execute the test suite, we discover that both Test2 and Test4 fail. To find the bug or bugs that produced these failures, we first investigate the failing test cases starting with Test2. Inspecting Test2's associated assertion, `cantAbandonAll`, we can see the body contains one line invoking the negation of the `crossRiver` predicate. To invoke the `crossRiver` predicate, four arguments have to be provided. All arguments are sets of objects with the following meanings:

- **from:** set of objects on the 'near' side of the river before the farmer crosses.

```
pred solvePuzzle{
    ord/last.far = Object
}

assert  cantAbandonAll{
    !crossRiver[Object, Fox+Chicken+Grain, none, Farmer]
}

assert noQuantumObjects {
    no s : State | some x : Object | x in s.near and x in s.far
}

assert farmerTakesAtMostOne{
    no s: State, s': ord/next[s] {
        {#{s.near - s'.near - Farmer} = 2 and no s.near.eats}
    }
}

assert farmerCantTakeFoxFirst{
    !crossRiver[Object, Grain+Chicken, none, Farmer+Fox]
}
```

Figure 6.1: Extension to Framers Model

- **from':** set of objects on the 'near' side of the river after the farmer crosses.
- **to:** set of objects on the 'far' side of the river before the farmer crosses.
- **to':** set of objects on the 'far' side of the river after the farmer crosses.

Where the 'near' side of the river is the side the farmer starts on for the state and 'far' is the opposite side.

Figure 6.2: Valuation of farmers model targeting solving the puzzle

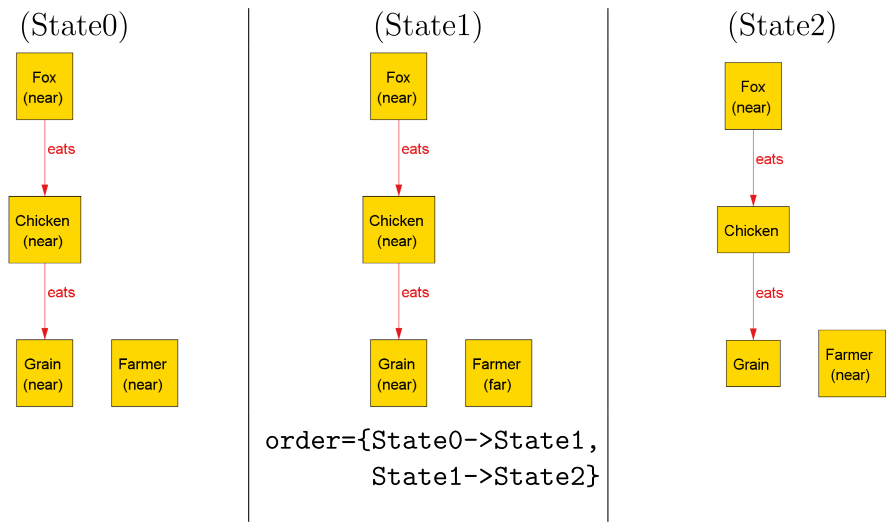

Figure 6.3: Valuation for farmers - farmer leaves everything

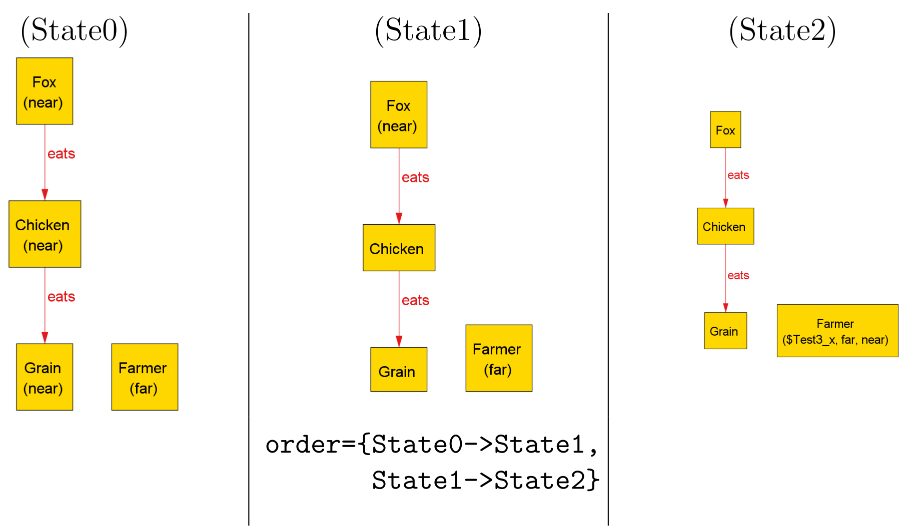

Figure 6.4: Valuation for farmers - farmer on both sides

From the `cantAbandonAll` assertion, the invocation of `crossRiver` specifies that at first, all the objects are on the near side of the river (**from:**{Object},

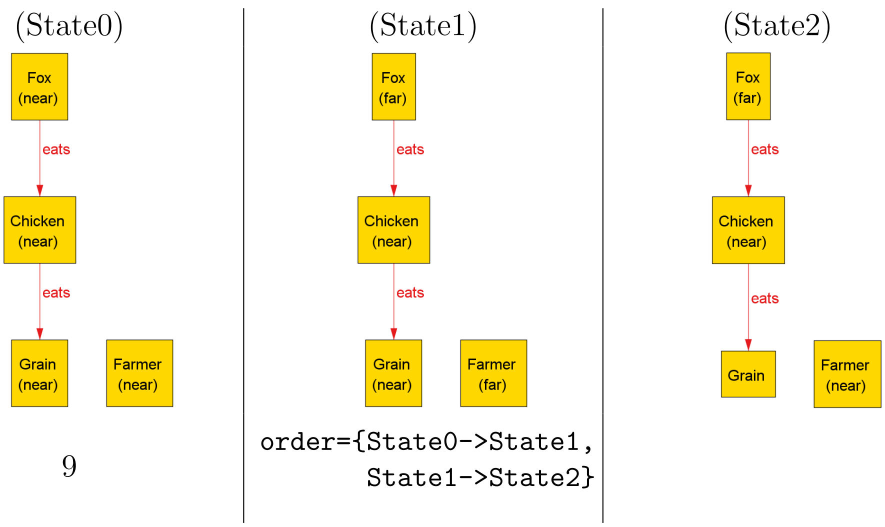

Figure 6.5: Valuation for farmers - taking fox first

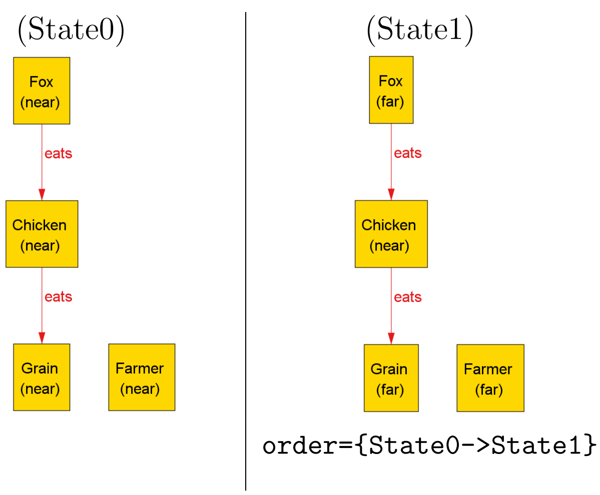

Figure 6.6: Valuation for farmers - taking two items

to:{none}). Then, the farmer crosses alone leaving the fox, chicken and grain behind (**from':**{fox+chicken+grain}, **to':**{farmer}). According to the rules of the logic problem, the fox should eat the chicken and the chicken should eat the grain. In other words, it is impossible for all 3 objects to be left on the same side. Therefore, the only valid `from':` set is {fox} if the farmer crosses alone at the start. Since Test2 forces the `from'` set to also contain the chicken and grain, the call to `crossRiver` should not hold. Therefore, since the assertion calls for the negation of `crossRiver`, the assertion should hold. Yet, the invocation of `crossRiver` ends up being valid, resulting in the `cantAbandonAll` assertion falsely being satisfiable. When we inspect our valuations, we have correctly captured the appropriate behavior (i.e. State2 in Figure 6.3 only the fox and farmer remain uneaten. We can hypothesize the error resides in the `crossRiver` predicate.

To support this idea, we turn to the behavior the second failing test case, Test4, which invokes the assertion `cantTakeFoxFirst`. Similar to Test2, the body of the assertion invokes the `crossRiver` predicate and taking the negation of `crossRiver`. Looking at the set arguments passed, initially all of the objects are on the near side of the river. After the farmer crosses, he takes just the fox with him, leaving the grain and chicken together. According to our eating rules, the chicken will eat the grain, resulting in the specified `from'` argument, {fox+chicken}, preventing `crossRiver` from holding. However, the call to `crossRiver` ends up being true, resulting in the `cantTakeFoxFirst` assertion incorrectly being satisfiable.

The common thread between the two failing test cases appears to be calling the `crossRiver` predicate in such a way that the behavior of the farmer should result in a failure. Specifically, both times, the `from'` set ends up

```
pred crossRiver [from, from', to, to': set Object] {
    (from' = from - Farmer - from'.eats and to' = to + Farmer) or
    (one x : from - Farmer | {
        from' = from - Farmer - x - from'.eats
        to' = to + Farmer + x
    })
}
```

Figure 6.7: New crossRiver predicate for the farmers model

violating the eating rule, but the `farmers` model fails to catch this erroneous behavior. Applying this insight, we can draft a new `crossRiver` predicate, outlined in Figure 6.7, in which we modify when the eating behavior is accounted for.

Utilizing the new `crossRiver` predicate, we can re-execute our test suite to determine if we have properly identified and resolved the bug. Now, all of the five test cases pass.

6.2 Scenario 2: Discovering a Bug in an Alloy Test Case

When a test case fails, a developer can infer that there might be a bug in one of two places: the Alloy model under test or the test case itself. Our first scenario highlighted how our framework can alert developers to bugs in the Alloy model. For this scenario, we will investigate how our framework can help the developer discover the bug lies within the test case.

There are three distinct ways that a developer can accidently introduce a bug into a test case:

- Specifying a valuation incorrectly
- Selecting the wrong command

To illustrate the process of debugging a test case, we will consider the `BinaryTree` model outlined in Figure A.2. Similar to the `List` model, the model allows any number of binary trees all of which adhere to an acyclicity constraint. In order to uncover a bug in a test case, we first need to draft a test suite for `BinaryTree`. To do so, we have three valid valuations, seen in Figure 6.8, and three invalid valuations, seen in Figure 6.9. For the three valuations in Figure 6.8, we intend for all to be instances of the `acyclic` predicate. Therefore, we can create the following three test cases:

Test1: ⟨Figure 6.8(a), "`run acyclic`"⟩
Test2: ⟨Figure 6.8(b), "`run acyclic`"⟩
Test3: ⟨Figure 6.8(c), "`run acyclic`"⟩

In addition, we have our three valuations from Figure 6.9. However, we created these valuations with the intention that they would not be valid. Therefore, we can create the follow three test cases in which we leave the command empty, since we did not intend for them to satisfy the `acyclic` constraint:

Test4: ⟨Figure 6.9(d), ϵ⟩
Test5: ⟨Figure 6.9(e), ϵ⟩
Test6: ⟨Figure 6.9(f), ϵ⟩

When we execute our test suite, we discover Test6 fails. To figure out where the bug may lie, we first follow the same initial steps as scenario 1 and look to the paragraphs the failing test cases invokes in its command. However,

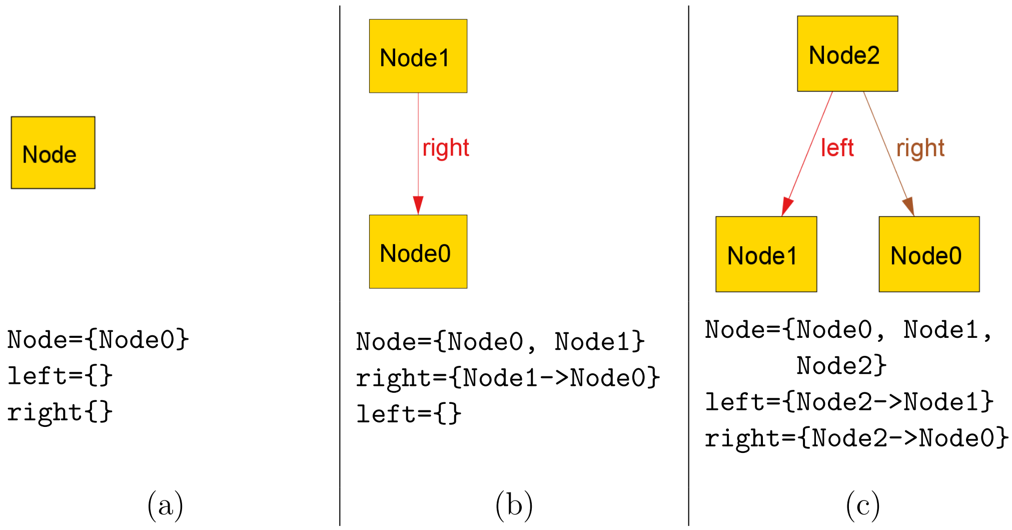

Figure 6.8: Valid Valuation for BinaryTree Model

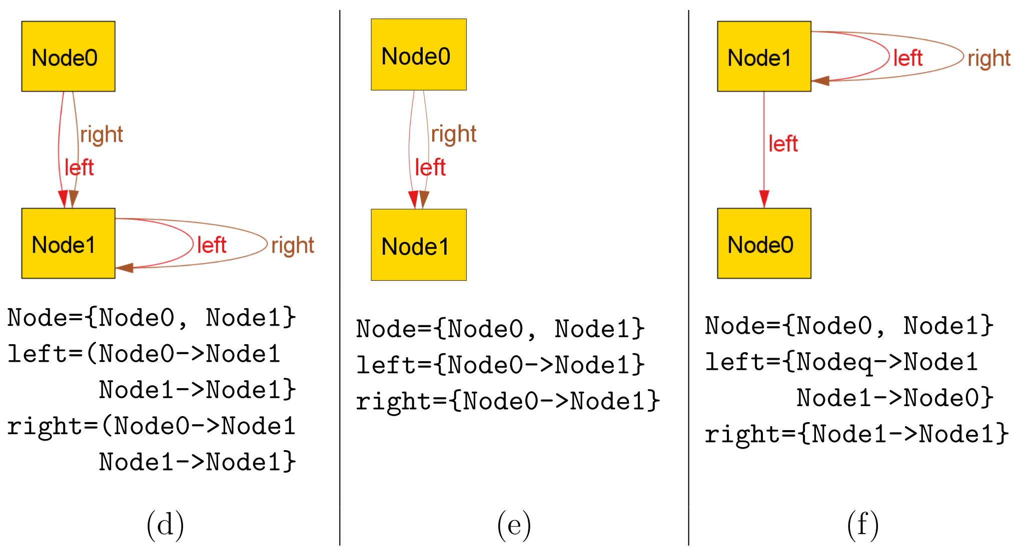

Figure 6.9: Invalid Valuation for BinaryTree Model

in this case, the command was left empty. Therefore, an empty predicate is invoked. There are two ways in which a valuation can fail to satisfy the empty command: the wrong scope was applied (i.e. the scope is less than needed for the valuations sets and relations) or the valuation fails to adhere to the constraints laid out in the facts of the model. The scope needed for the valuation depicted in Figure 6.9(f) is "`for 2 Node` or greater. Therefore, simply running an empty predicate with the default scope will be ok. As a result, we can conclude the failure is related to the only *fact* in `BinaryTree`. The fact requires any `Nodes` `left` relation and `right` relation to adhere to the `lone` multiplicity. In the failing test case, the `Node1s` `left` relation is two, violating the fact. Consequently, the test case fails.

The `BinaryTree` model is correct. We do want the two relations, `left` and `right`, to be restricted to either no mapping or one map. Therefore, the fault lies within our test case and leaving the command empty. When drafting the second set of test cases, all three commands could have been "`run {!acyclic}`" instead of the commands being left empty. However, this would still result in Test4 and Test5 passing while Test6 still fails. Since the fact is always applied, we need to structure our command to account for this behavior. The two commands, `run` and `check`, have different default expectations. A `run` command is expected to be satisfiable whereas a `check` command is expected to be unsatisfiable. Therefore we could rewrite our test case to be the following: ⟨Figure 6.9(f), "`check {}`"⟩. When we execute this newly modified test suite, the test suite passes. To be closer to our original intent, it could be considered good practice to restructure Test4 and Test5 to have the "`run {!acyclic}` command.

6.3 Scenario 3: Adding a Test to Improve Coverage

In an imperative language, a developer may execute a branch coverage tool only to discover the associated test suite repeatedly takes the same choice at a branch, leaving a section of code completely uncovered. Similarly, an Alloy developer may execute a formula coverage tool only to reveal a particular formula f repeatedly produces the same evaluation value or even worse, fails to reach f. In order to improve coverage, the developer in either situation only has one thing to do: add new test case(s).

In the second scenario, we look at fixing errors in a test suite for a binary tree. Building off of the binary tree example, we can create a `FullTree` model, a binary tree in which all nodes except for the leaves have both a left and right child, captured in Figure A.3.

Since the `FullTree` model is implemented by building off of the `BinaryTree` model, we can incorporate our existing `BinaryTree` test suite as a good starting point. Therefore, we use the same cases six test updated to reflect the corrections from scenario 2 and the new predicates of `FullTree`:

Test1: ⟨Figure 6.8(a), "`run FullTreeOk`"⟩
Test2: ⟨Figure 6.8(b), "`run acyclic`"⟩
Test3: ⟨Figure 6.8(c), "`run FullTreeOk`"⟩
Test4: ⟨Figure 6.9(d), "`run {!acyclic}`"
Test5: ⟨Figure 6.9(e), "`run {!acyclic}`"
Test6: ⟨Figure 6.9(f), "`check {}`"⟩

Then, we can execute the test suite over our `FullTree` model in order to see where, if anywhere, we need to improve coverage. As we can see in

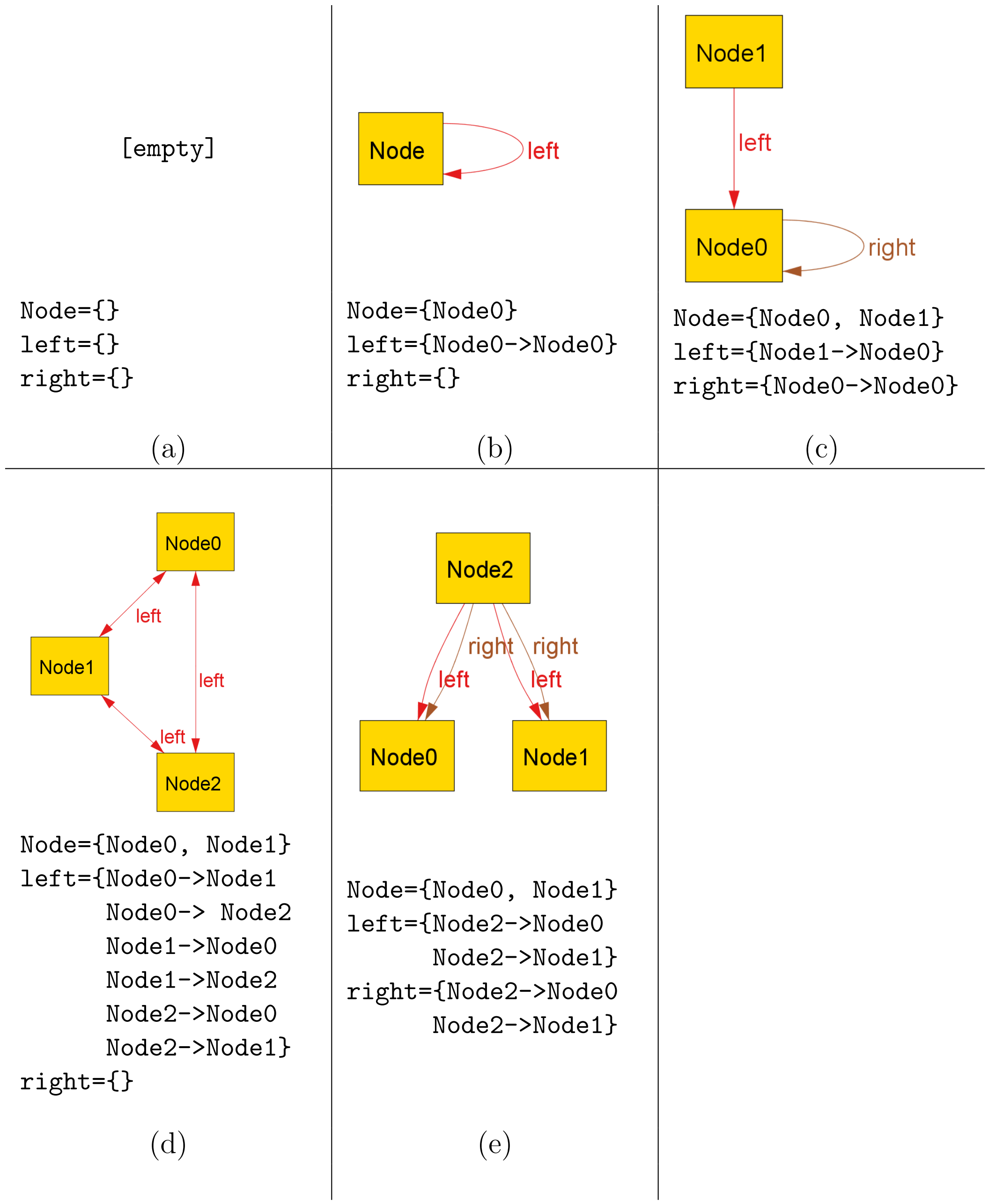

Figure 6.10: Valuations for FullTree Model

Coverage Metric	# Req. Covered	# of Feasible Req.	Coverage
Signature Coverage	2	3	66.6%
Relation Coverage	3	3	100%
Expression Coverage	31	42	73.81%
Fact Coverage	6	14	42.86%
Predicate Coverage	20	26	76.90%
Formula Coverage	26	42	61.90%

Figure 6.11: Coverage for Initial Test Suite

Figure 6.11, there are gaps in coverage. In particular, only relation coverage is completely covered. At the opposite end of the spectrum, fact coverage is significantly worse than all other coverage metrics. However, with the exclusion of relation coverage, all coverage metrics have serious room for improvement.

When we look into a full diagnostic, we can see what requirements for the various metrics are not covered. A glaring absence is the coverage provided by a valuation where all sets and relations are equivalent to the empty set. Additionally, the only fact of `FullTree` never once evaluated to false. Since a quantified formula is present in the fact, this impact is greatly felt. Therefore, another good point for improving coverage is to draft valuations that cover these false values for all the formulas in the fact. Based on these points of weakness and the other gaps in coverage, we draft a series of five new test cases using the valuations in Figure 6.10:

Coverage Metric	# Req. Covered	# of Feasible Req.	Coverage
Signature Coverage	3	3	100%
Relation Coverage	3	3	100%
Expression Coverage	42	42	100%
Fact Coverage	13	14-1	100%
Predicate Coverage	26	26	100%
Formula Coverage	41	42-1	100%

Figure 6.12: Coverage for Extended Test Suite

Test7: ⟨Figure 6.10(a), "`run FullTreeOk`"⟩
Test8: ⟨Figure 6.10(b), ϵ⟩
Test9: ⟨Figure 6.10(c), ϵ⟩
Test10: ⟨Figure 6.10(d), "`check {}`"
Test11: ⟨Figure 6.10(e), "`check {}`"

When we apply all of this together, we can execute our new test suite and determine if we need to continue to add test cases to improve coverage. This time, all of the coverage metrics are at 100 percent. Of note, both fact coverage and formula coverage involved an infeasible requirement, which is why the number of requirements is listed in the form "X-1." The formula "`all n : Node| lone n.left && lone n.right`" has yet to satisfy the criteria "$|d| = 1$, b is false". Similar to the problem with the universal formula in `List`'s fact `PartialFunction`, this requirement ends up being infeasible. Therefore, we have covered all the feasible requirements and this we have reached full coverage.

Chapter 7

Conclusion and Future Work

We introduced some central ideas to lay the foundation of AUnit, our test automation framework for Alloy envisioned in the spirit of the xUnit frameworks for imperative programs. Our goal was to ease the burden of developers by providing an intuitive methodology for testing models. One of our key contribution is to define the concepts of declarative test case, thus laying the groundwork for exploring a range of testing techniques within the scope of Alloy. Through the use of complete and partial valuations, our test case format enables a user to explore a wide range of behaviors with their test suite. While complete valuations based test cases can be executed by simply invoking the `evaluator`, partial valuation based test cases require the additional use of a SAT solver. However, allowing for both leads to a robust testing framework.

To expand on our testing infrastructure, we have developed the notion of test coverage as well as a family of coverage criteria for Alloy models. The coverage criteria is centered around two core Alloy constructs: expressions and formulas. Both have multiple criteria that need to be met by some set of test cases in order for a given expression or formula to be considered *covered*.

We present eight different coverage metrics with model coverage subsuming everything. In addition, our model coverage metrics provide a novel basis for *scenario exploration* [14]. In order to calculate coverage, we focus on mapping the behavior of a test cases valuation to the criteria the valuation covers. At times, we may run into infeasible criteria. Altogether, our definitions of declarative tests and test coverage serve to meet our design goals in the following ways:

- From experience, switching from an imperative viewpoint to a declarative viewpoint can be difficult. With the introduction of valuations into the test case format, a user has a concrete viewpoint of if the model behaves as expected. When a failure arises, the tester now has a narrowed starting point to help isolate where the bug is within the model. This is especially important considering just one symbol can lead to a faulty model, i.e. using '^' instead of '*'.
- With a coverage framework included, testers can draft comprehensive test suites that enable testers to feel more confident with their model as the suite gets closer to full coverage. With the wide array of coverage metrics, we are able to get a good sense of what aspects of the model are being tested. Through targeting the gaps in coverage, we can focus on drafting test cases which will test new areas of the model instead of repeating behavior.

Currently, we are working to implement AUnit as an extension that integrates into the standard Alloy tool-set and supports both writing tests and reporting coverage. The goal is to model the style of reporting similar to JUnit for a test suite. For coverage, the information will be reported by coloring (partially) covered expressions and formulas (in the spirit of code coverage tools for imperative programs [2]) for the model displayed on the left hand side of the tool. An interesting situation occurs for processing the coverage of test cases that involve a partial valuation. The solution ends up being relatively straightforward. As a user enumerates instances, those instances will update the coverage calculations.

Our work opens the possibility of adapting for Alloy several well-known testing techniques that have shown to be effective in the context of imperative programs. For example, our coverage criteria could provide a basis for introducing *directed test generation* [5] for Alloy. More broadly, techniques for *regression testing* [17] can now be considered for Alloy. Moreover, while the basic inspiration of AUnit is to facilitate testing of Alloy models, we believe the analogies between declarative programming and imperative programming, which lie at the heart of AUnit, also provide the basis of a more comprehensive framework for development and maintenance of Alloy models.

Appendices

Appendix A

Alloy Model Appendix

A.1 Farmers Alloy Model

The `Farmers` model introduces a number of new Alloy syntax. First, `Farmers` applies new constraints to the signatures within the model. In the model, Chicken, Fox, Grain and Farmers `extend` Object, meaning all four are disjoint subsets of Object. Alloy supports `abstract` signatures. Similar to abstract classes, these signatures have no elements except those belonging to extending signatures. In addition, we can see that the signature declarations for Farmer, Fox, Chicken, and Grain are all restricted by the multiplicity `one`. Alloy supports three different multiplicity constraint values for signature declarations: exactly one (`one`), less than or equal to one (`lone`), and greater than or equal to one (`lone`). These multiplicity constraints plus an addition one, any number (`set`), can in turn be applied to a wide range of Alloy features.

Fact `eating` contains a cross product ('`->`') and a union set operator ('`+`'). In addition, other set operators used in this model as well as some of the other examples includes: intersection ('`&`'), difference ('`-`'), and equality ('`=`'). It is important to note that in Alloy, the '`=`' symbol represents equality and not assignment.

```
module Farmers

open util/ordering[State] as ord

abstract sig Object { eats: set Object }
one sig Farmer, Fox, Chicken, Grain extends Object {}

fact eating { eats = Fox->Chicken + Chicken->Grain }

sig State {
    near: set Object,
    far: set Object
}

fact initialState {
    let s0 = ord/first | s0.near = Object && no s0.far
}

pred crossRiver [from, from', to, to': set Object] {
    ( from' = from - Farmer && to' = to - to.eats + Farmer ) ||
    (some item: from - Farmer {
        from' = from - Farmer - item
        to' = to - to.eats + Farmer + item
    })
}

fact stateTransition {
    all s: State, s': ord/next[s] {
        Farmer in s.near =>
            crossRiver[s.near, s'.near, s.far, s'.far] else
            crossRiver[s.far, s'.far, s.near, s'.near]
    }
}
```

Figure A.1: Farmers Alloy Model

```
module BinaryTree

sig Node{
    left: set Node,
    right: set Node
}
fact{ all n : Node| lone n.left && lone n.right }

pred acyclic{
    all n : Node{
        n !in n.^(left + right)
        lone n.~(left + right)
        no n.left & n.right
    }
}
```

Figure A.2: Binary Tree Alloy Model

The `crossRiver` predicate introduces both the conjunction logical operator (‘`&&`’) while `stateTransition` contains implication (‘`=>`’). Additional logical operators supported by Alloy include: disjunction (‘`||`’), bi-implication (‘`<=>`’), negation (‘`!`’)

A.2 BinaryTree Alloy Model

The `BinaryTree` model primarily introduces one new Alloy feature: transpose (~). Transpose in the context of Alloy refers to creating a new relation by flipping the order of atoms in the relation. For our `BinaryTree`, we apply the transpose to the left and right children of a node, meaning we have created a relation that relates a child node to its parent node. The rest of the Alloy language presented has been used in either the `List` or the `Farmers` models.

A.3 FullTree Alloy Model

The `FullTree` model builds on top of the `BinaryTree` model. However, it does introduce and new Alloy feature in the `makeFull` predicate: set cardinality (‘`#`’).

A.4 Subsumption Relationship Model

The model in Figure A.4 is the Alloy model which was used to generate the subsumption relationship graph.

```
module FullTree

sig Node{
    left: set Node,
    right: set Node
}
fact{ all n : Node| lone n.left && lone n.right }

pred acyclic{
    all n : Node{
        n !in n.^(left + right)
        lone n.~(left + right)
        no n.left & n.right
    }
}

pred makeFull{
    all n : Node | #{n.*left} = #{n.*right}
}

pred FullTreeOk{
    acyclic[]
    makeFull[]
}
```

Figure A.3: FullTree Alloy Model

```
module subsumption

abstract sig Criteria { subsumes: set Criteria }

one sig SC, RC, EC, FaC, PC, AC, FC, MC extends Criteria {}

fact {
    MC.subsumes = EC + FC
    EC.subsumes = RC
    RC.subsumes = SC
    FC.subsumes = FaC + PC + AC
    no (SC + FaC + PC + AC).subsumes
}

pred subsumption() {}
run subsumption
```

Figure A.4: Coverage Metric Subsumption Alloy Model

Bibliography

[1] Alloy a language and tool for relational models. `http://alloy.mit.edu/alloy/`.

[2] EclEmma code coverage tool. `http://www.eclemma.org`.

[3] JUnit test automation framework. `http://junit.org/`.

[4] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[5] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *ICSE*, 2011.

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[7] Niklas Een and Niklas Sorensson. An extensible SAT-solver. In *SAT*, 2003.

[8] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[9] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012.

[10] Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints.* PhD thesis, MIT EECS, December 2003.

[11] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Santa Margherita Ligure, Italy, May 2003.

[12] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, 2001.

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

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

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

[16] Emina Torlak and Greg Dennis. Kodkod for alloy users. In *First Alloy Workshop*, 2006.

[17] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2), 2012.

Vita

Allison Kathleen Sullivan was born in Houston, Texas. She received her Bachelors of Science degree in Software Engineering for The University of Texas at Dallas. She graduated with both Summa Cum Laude and Erik Johnson School of Engineering departmental honors. In 2012, she applied to The University of Texas at Austin graduate program and was admitted. She started her graduate studies in Software Engineering in September 2012.

Email address: allisonksullivan@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.