Towards a Test Automation Framework for Alloy

Allison Sullivan Razieh Nokhbeh Zaeem Sarfraz Khurshid Darko Marinov
University of Texas University of Illinois
Austin, TX 78712 Urbana, IL 61801
USA USA
{allinsonsullivan,nokhbeh,khurshid}@utexas.edu marinov@illinois.edu

ABSTRACT

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, testing techniques that are effective in practice in the context of non-procedural or declarative models. Our primary design goals are: (1) to facilitate writing Alloy models correctly for users who are new to Alloy; and (2) to enable more effective 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).

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging;
D.2.2 [Software Engineering]: Design Tools and Techniques

General Terms
Verification

Keywords
Test case, code coverage, coverage criteria, Alloy, JUnit

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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 [7] – à la model checking [5] – 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. Two factors make this task particularly demanding. One, 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. Two, toolsets 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 [7], 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 [6] 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 paper 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 [2] that are used widely in practice for automated testing, albeit largely in the context of non-declarative programs. Our primary design goals are: (1) to facilitate writing Alloy models correctly for users who are adept at commonly used programming languages but maybe new to Alloy; and (2) 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 cover-
age (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 \((\sigma, \rho)\) where \(\sigma\) is an assignment of values to the relations in the model and \(\rho\) is an Alloy command that defines the constraint-solving problem. A test passes if \(\sigma\) is a solution with respect to the command \(\rho\), and fails otherwise. Our definition of model coverage blends the spirit of logic-based coverage for imperative programs (e.g., clause coverage or predicate coverage [3]) 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 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 evaluation where Alloy formulas and expressions are evaluated for each given assignment to determine test pass/fail results and coverage requirements that are met. Thus, our proposal does not require SAT solving, which has much higher complexity than evaluation.

We make the following contributions: (1) 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; (2) declarative test cases – we formalize the definition of test cases for Alloy models and define the semantics of passing and failing of tests; (3) model coverage – we introduce eight criteria for computing model coverage and present a subsumption relation among the coverage criteria; and (4) example – we present an illustrative demonstration of declarative tests and model coverage using a small yet complex Alloy model.

2. EXAMPLE

Figure 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 sig Node declaration introduces Node as a set of atoms and link as a binary relation that has the type Node \(\times\) Node. The fact (fact) PartialFunction specifies that each node is related to at most one node (lone) under the link relation, i.e., link is a partial function. The predicate (pred) NoDirectedCycles uses universal quantification (all) to define acyclicity. The operator \(\sim\) 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 illustrates three example instances generated for this command by the analyzer. Figure 3 illustrates three example valuations that are not instances and will not be generated for this command by the analyzer.

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. Signature declarations may introduce facts implicitly.

Predicate (pred). A pred is a named (and optionally parameterized) formula, which can be invoked elsewhere. Alloy Analyzer does not allow recursive predicates and inlines all predicate invocations before solving them.

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 pred-
icate 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. 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.

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, 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 ≥ 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 with a command, the analyzer executes the command 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. A model may have more than one commands and the user may select a specific command or all of them to execute. The analyzer adds symmetry-breaking predicates to remove isomorphic solutions and reduce the total number of solutions [10].

4. FOUNDATIONS: AUnit

We represent an Alloy model as a quintuple ⟨S, F, P, A, C⟩, 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 = ⟨S, F, P, A, C⟩ be an Alloy model. Assume S is non-empty. Let Ξ be the set of all expressions (other than variable declarations or uses) in the parse tree of m. Let Φ be the set of all formulas in the parse tree of m. The sizes of Ξ and Φ are linear in the size of m.

For ρ ∈ C, let Ξρ ⊆ Ξ and Φρ ⊆ Φ be the expressions and formulas respectively in the constraint-solving problem for ρ. E.g., for the list model: Ξ = {Node, link, “link. n.link”, “all n: Node | n !in n.^link”, “all n: Node | n !in n.^link”, “all n: Node | n !in n.^link”}; Ξ_{run NoDirectedCycles} = Ξ: Φ = {“all n: Node | lone n.link”, “all n: Node | lone n.link”, “all n: Node | lone n.link”, “all n: Node | lone n.link”, “all n: Node | lone n.link”, “all n: Node | lone n.link”}; and Φ_{run NoDirectedCycles} = Φ.

Let ΞF, ΞP, and ΞA (each ⊆ Ξ) respectively be the sets of all expressions that appear in any fact, predicate, or assertion. Let ΦF, ΦP, and ΦA (each ⊆ Φ) respectively be the sets of all formulas that appear in any fact, predicate, or assertion.

4.1 Declarative Test Cases

Definition 1. A test case t = (σ, ρ) where σ is an assignment of values to all sets and relations in S, and ρ is either the default command “run {}” or a command that invokes a predicate in P or an assertion in A.

Thus, a test case may have commands other than those that already exist in the model, i.e., belong to set C.

Definition 2. A test case t = (σ, ρ) passes if σ is a solution to the constraint-solving problem for the command ρ and otherwise, t fails.

To illustrate, let σ1 be any instance in Figure 2; then, the test case ⟨σ1, “run NoDirectedCycles”⟩ passes. As another example, let σ0 be the valuation in Figure 3(μ); then, the test case ⟨σ0, “run NoDirectedCycles”⟩ fails since σ0 is not an instance of the “run NoDirectedCycles” command; note however, the test case ⟨σ0, “run {NoDirectedCycles}”⟩ passes.

4.2 Coverage Computation

Let T be a test suite.

4.2.1 Coverage: Test Case

Let t = (σ, ρ) ∈ T be a test case.

Definition 3. The coverage obtained for t is a pair of maps (πt, ωt) where:

- πt maps each Alloy expression in Ξρ to the set(s) of tuples it evaluates to for assignment σ; and
- ωt maps each Alloy formula in Φρ to the boolean value(s) it evaluates to for assignment σ.

To illustrate, let σ be the instance shown in Figure 2(β) and ρ = “run NoDirectedCycles”. Then π(σ,ρ) is:

Node0 = {Node0, Node1},
link0 = {Node0->Node0},
"link. n.link" = true,
"all n: Node | n !in n.^link" = true,
"lone n.link" = true,
"n !in n.^link" = true

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

Moreover, ω(σ,ρ) 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.

4.2.2 Coverage: Test Suite

Definition 4. The coverage obtained for test suite T is a pair of maps (πT, ωT) where:

- πT = ∪t∈T πt,
- ωT = ∪t∈T ωt.

4.3 Coverage Criteria

The basis of our model coverage criteria are four sets of coverage requirements – three (R0, R1, and R2) based on Alloy expressions and one (R3) based on Alloy formulas:

- R0 – 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| ≥ 2.
- For the list example, R0 has a total of 3 requirements (as there is only one set Node). The suite {(α, ε), (β, ε), (μ, ε)} covers R0.
- R1 – 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| ≥ 2.
- For the list example, R1 has a total of 3 requirements (due to the relation link). The suite {(α, ε), (β, ε), (γ, ε)} covers R1.
Definition 7. Expression coverage (EC):

Definition 6. Relation coverage (RC):

Definition 5. Signature coverage (SC):

Definition 9. Pred coverage (PC):

Definition 10. Assert coverage (AC): $R_3$ restricted to $\Phi_A$.

Definition 11. Formula coverage (FC): $R_3$ restricted to $\Phi_F$.

Definition 12. Model coverage (MC): $EC \cup FC$

4.3.1 Criteria Subsumption

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 4).

5. FUTURE WORK AND CONCLUSIONS

This paper introduced our vision of AUnit, a test automation framework for Alloy in the spirit of the xUnit frameworks for imperative programs. Our key contribution is to define the concepts of declarative test case and coverage, and a family of coverage criteria for Alloy models. We are currently implementing AUnit as an extension to the standard Alloy tool-set that supports writing tests and reports coverage obtained by coloring (partially) covered expressions and formulas (akin to code coverage tools for imperative programs [1]). We plan to allow users to provide partial solutions [8] to reduce the burden of test formulation and utilize SAT solving in test exploration and coverage computation. Our model coverage metrics provide a novel basis for scenario exploration [9]. We plan to study the effectiveness of AUnit is finding and removing bugs in Alloy models.

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 [4] for Alloy. More broadly, techniques for regression testing [11] 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.

6. ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation Grant Nos. CCF-0845628 and CCF-1012759.

7. REFERENCES