Automated Test Generation and Mutation Testing for Alloy

Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid
The University of Texas at Austin, USA
Email: {allisonksullivan,kaiyuanw,nokhbeh,khurshid}@utexas.edu

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

I. Introduction

Software designs play a vital role in developing dependable systems. While a number of notations exist for modeling designs [1], [3], [47], the tool support, e.g., integrated development environments or test automation tools, for building *correct* models remains inadequate, requiring developers to employ validation methodologies that can be unfamiliar, ad hoc, or not well-suited.

This paper focuses on testing models written in the well-known software modeling language Alloy [1]. Alloy is a declarative, first-order logic with transitive closure based on relations. Alloy offers expressive operators that allow succinct formulation of complex properties. However, the expressiveness and succinctness can make Alloy models look deceptively simple. Formulating them correctly and reasoning their correctness can be quite challenging, especially for non-experts.

The Alloy analyzer [24], [51] is an automatic tool for *scope-bounded* reasoning where the analysis results hold for the given *scope*, i.e., bound on the universe of discourse. Alloy users write *commands*, which take two forms: (1) *simulation*, where the analyzer finds an *instance*, i.e., a valuation to the relations in the model such that the formula evaluates to true; and (2) *checking*, where the analyzer finds a *counterexample*, i.e., a valuation such that the negation of the formula evaluates to true. Technically, the underlying analysis for both forms is the same – solving *logical constraints*. The analyzer translates the Alloy model to a *propositional formula* with respect to the scope and uses off-the-shelf SAT solvers to solve it, and translates SAT solutions to Alloy instances or counterexamples.

Models written in Alloy have two basic kinds of faults – *underconstraint*, where the formula allows valuations that the user wanted to rule out; and *overconstraint*, where the formula rules out valuations that the user wanted to allow. Alloy users employ two basic methods to validate their models. One, they use simulation to enumerate and inspect valuations to detect if some expected ones are missing or some invalid ones are present. Two, they use checking to validate expected properties between different formulas, e.g., checking (bounded) equivalence between two definitions that they expect are equivalent. Moreover, for unsatisfiable formulas, the users can also visually inspect *unsat cores*, which highlight parts of the formulas that make the analysis problem unsatisfiable.

While the Alloy analyzer provides critical functionality for checking Alloy models – which indeed is a key strength of the Alloy tool-set – validating the correctness of an Alloy model is conceptually very different from the widely used practice of *testing* imperative programs, which is conceptually simple: create some inputs (with respect to some coverage criterion or otherwise), run the program against them, and check the outputs. For example, there is no notion of unit tests or code coverage built into the Alloy tool-set. This lack of support for foundational elements in testing makes it particularly hard for Alloy users, especially beginners, to gain confidence in their models. Indeed, one question that we have continually faced for over 10 years of teaching Alloy is "how do I *test* my Alloy program?" Even for expert users, validating correctness can be complicated. For example, when checking a logical relation, say implication between formulas f and g, i.e., $f \Rightarrow g$, it is standard practice to increase the scope and re-run the analyzer to increase confidence that the implication indeed holds; however, if f is overconstrained and simply false, increasing the scope only leads to a false increase in confidence. Similarly, when simulating $f \wedge g$, and finding a solution, the user may inadvertently fail to notice that g is true but only vacuously.

Our previous work [50] introduced basic definitions for unit tests, test execution, and model coverage for Alloy to lay the foundation of testing Alloy models in the traditional spirit of testing. A *unit test* t for Alloy is a pair $\langle v, c \rangle$ where v is a valuation and c is a command. Test t *passes* if v is a valuation allowed by the Alloy analyzer on executing the command c, and *fails* otherwise; thus, test execution is a constraint

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.31

checking (and not solving) problem. Conceptually, a test case is a valuation with the label *valid* or *invalid*; thus, the Alloy users can create tests to directly check for underconstraint (when a given invalid valuation is allowed by the formula) and overconstraint (when a given valid valuation is not allowed by the formula). *Model coverage* follows the traditional spirit of code coverage and defines test requirements based on the different elements of the Alloy model. For example, for an Alloy expression e, the test requirements include e evaluating to the empty set, to a singleton set, and a non-empty non-singleton set for different valuations. As another example, for a universally quantified formula u, the test requirements include the case when u is *vacuously* true, i.e., the domain (for quantification) is the empty set, and cases when u is non-vacuously true (and separately false), e.g., when the domain has exactly 1 element, and the domain has > 1 element. Thus, the coverage requirements require the tests to *exercise* various different values Alloy expressions and formulas may take as well as the different ways those values may arise.

This paper introduces two new approaches for automated testing of Alloy models. Our first approach provides automated test generation and is embodied by three techniques that create test suites following the spirit of traditional black-box, white-box, and mutation-based testing. Our second approach brings the spirit of traditional mutation testing to Alloy and defines how to create mutants of Alloy models, compute mutation testing results, and check for equivalent mutants using SAT. The black-box test generation technique *AGen*$_{BB}$ creates suites that include all (non-isomorphic) instances in the given scope and brings the spirit of *bounded exhaustive testing* for imperative programs [7], [35] to Alloy models. The white-box test generation technique *AGen*$_{Cov}$ brings the spirit of *coverage-directed input generation* for imperative programs [8], [18], [43] to Alloy models. *AGen*$_{Cov}$ reduces the problem of directed test generation for Alloy to constraint solving where model coverage requirements (introduced by AUnit) are part of the constraint. *AGen*$_{Cov}$ iteratively builds a minimal set of (non-isomorphic) tests to meet the chosen criterion. The mutation-based generation technique *AGen*$_{Mu}$ also uses directed test generation but bases it on mutant killing [55]. All three techniques use Alloy's SAT-based back-end for test generation and can be adapted to create suites based on different solving *strategies* [33], [39].

This paper makes the following contributions: (1) **Automated test generation for declarative models** – we introduce automated techniques to generate tests for Alloy models to support black-box, white-box, and mutation-based test generation; (2) **Mutation testing for declarative models** – we introduce techniques to perform mutant generation, mutation score calculation, and equivalent mutant detection (when feasible) for Alloy models; (3) **Implementation** – we implement a prototype that embodies our techniques as well as the theoretical foundations introduced previously by AUnit, specifically to provide test case execution and code coverage computation in addition to automated test generation; and (4) **Experiments** – we perform a two-fold evaluation to show the efficacy of our approach. One, we compute model coverage and mutation score (which is often considered as the strongest test adequacy criterion in imperative programs) for the generated tests. Two, we show our approach finds real faults in several subjects, including all 19 faulty models submitted as solutions to a homework question in a graduate course in our Department.

II. Illustrative Example

This section presents a small but representative example of a faulty Alloy model to introduce some key concepts in Alloy, AUnit, and our test generation techniques. First, we will step through how to create our example Alloy model, then we will demonstrate how AUnit tests can be used to detect the bug in the model, and finally we will illustrate a mutant Alloy model. Figure 1(a) (incorrectly) models an *acyclic* singly-linked list. The *signature* (`sig`) declaration "`sig List`" introduces a set of list atoms; similarly "`sig Node`" introduces a set of node atoms. The keyword "`one`" declares `List` to be a singleton set. Each signature declaration also introduces a binary relation. The relation `head` maps lists to nodes. The relation `link` maps nodes to nodes. Both `head` and `link` are partial functions as declared by the keyword `lone`. The *predicate* (`pred`) `Acyclic` defines acyclicity. The predicate body contains an implication ("`=>`") and intends to state that if the list has some head node, there exists a node reachable from the head with no link, i.e., the list is "null-terminated". The formula "`some E`" for expression `E` states that `E` is a non-empty set. The keyword "`some`" also represents existential quantification. The operator '`.`' is relational composition and '`^`' is transitive closure. The expression "`l.head.^link`" represents the set of nodes reachable from `l`'s head following 1+ traversals along `link`. Structuring `Acyclic` as an implication additionally allows for the list to be empty, as without the implication, the existentially quantified formula requires there to be at least one Node atom in the List.

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

To explore this fault, we can look at our AUnit tests. The predicates `Val1`, `Val2`, and `Val3` together with their respective labeled `run` commands `Test1`, `Test2`, and `Test3` represent three AUnit tests. Intuitively each predicate represents a test *input* and each command represents a test *oracle*. `Test1` and `Test3` pass, but `Test2` fails and exposes a fault in the model. The valuation `Val2` is expected to be a valid

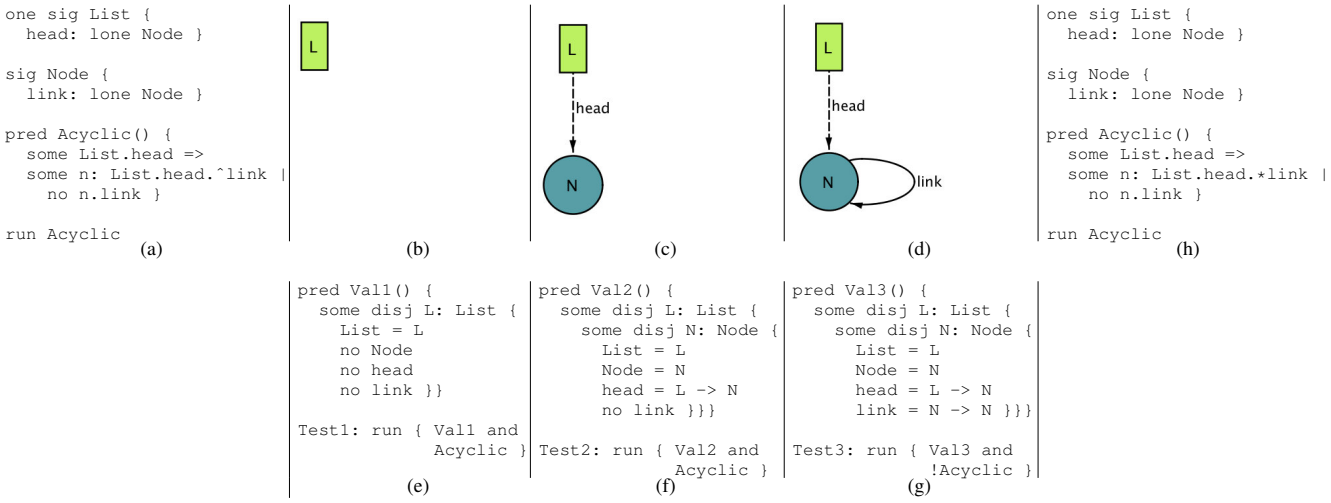

Fig. 1: (a) Singly-linked acyclic list model – faulty. (b) Instance: empty list. (c) Non-instance: acyclic list with 1 node. (d) Non-instance: cyclic list with 1 node. (e) Passing test with valuation (b). (f) Failing test with valuation (c). (g) Passing test with valuation (d). (h) Mutant of model (a) – transitive closure operator (‘^’) is replaced with reflexive transitive closure (‘$*$’).

acyclic list but the predicate `Acyclic` is overconstrained and erroneously disallows it; the fault is in the use of transitive closure (‘^’) instead of *reflexive transitive closure* (‘$*$’). Note that `Val3` is not acyclic and `Test3` passes since the test oracle correctly identifies it (`!Acyclic`).

For the Alloy model in Figure 1(a) and the default scope of 3, our black-box generation technique $AGen_{BB}$ creates 157 inputs, our coverage-driven test generation technique $AGen_{Cov}$ creates 10 inputs, and our mutation-based technique $AGen_{Mu}$ creates 10 inputs. Each of $AGen_{BB}$, $AGen_{Cov}$, and $AGen_{Mu}$ produces all 3 valuations shown in Figure 1.

Figure 1(h) shows an example mutant created by our mutant generator for the faulty model (Figure 1(a)), where the operator ‘^’ is replaced with the operator ‘$*$’. This mutant is killed by test case (f), which fails against the faulty model (Figure 1(a)) but passes against this mutant. Therefore, the mutant Alloy model in Figure 1(h) addtionally represents one possible solution for fixing the original faulty model.

III. Automated Test Generation Using AUnit

This section presents our automated test generation approach. We introduce three basic techniques. $AGen_{BB}$ is a *black-box* technique which uses constraint solving to enumerate solutions for commands that execute (or check) existing predicates (or assertions) but does not directly utilize the way the Alloy model is written (Section III-A). $AGen_{Cov}$ is a *white-box* technique, which performs targeted test generation driven by AUnit's model coverage requirements (Section III-B). $AGen_{Mu}$ is a mutation-based technique, which we describe after we introduce mutation testing for Alloy in Section IV.

A. Enumeration-based Input Generation

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

Specifically, for each instance i enumerated by executing ϵ, we add a test case per each predicate and assertion paragraph present in the model, using the negation of the paragraph if the valuation fails the paragraph (the `i.eval(P)` call is false). Since all these valuation must satisfy the facts of the model, to reduce the number of overall tests, we do not explicitly generate tests that have ϵ as their command. Taking this into account, our new test suite for the model in Figure 1(a) is: $\langle\langle$Figure 1(b), run Acyclic$\rangle$, $\langle$Figure 1(c), run Acyclic$\rangle$, $\langle$Figure 1(d), run !Acyclic$\rangle\rangle$.

While we now have a more robust test suite, all the tests have a common limitation: we never generate any tests in which the facts of the model are violated. Therefore, we create a second Alloy model, M', in which all the facts in the model are replaced with the disjunction of the negation of all the facts (`negateFacts`). Any instance satisfying M'

Algorithm 1: *AGen*$_{BB}$ – Enumeration-based Input Generation

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

violates at least one `fact` formula. If we repeat the process with M' (run ϵ, enumerate all instances, build up a test suite) we end up with extremely large state spaces, resulting in an unreasonable number of valuations to consider. Instead, we focus on enumerating one instance per each size in the scope. If ϵ is satisfiable for the tailored scope, a test case of the form $\langle$first instance enumerated, `run` ϵ `expect 0`$\rangle$ is added to the test suite. An `expect` clause is an Alloy construct used to state whether a command should be satisfiable (`expect 1`) or not (`expect 0`). By appending `expect 0` to the empty command, the tester is informed that the associated valuations do not adhere to the facts of the model.

We implemented Algorithm 1 to embody *AGen*$_{BB}$ using the Alloy Analyzer, which is the enumerator associated with the standard Alloy distribution. Although the SAT solver needs to be invoked when we first execute ϵ and enumerate instances, we can actually build up the remainder of the test suite without invoking the SAT solver: by using the Kodkod tool's `evaluator` API [51]. We invoke the `evaluator` by passing a test's valuation and an Alloy `formula` for a predicate or assertion. The `evaluator` returns `true` if the instance satisfies the `formula` and `false` otherwise, indicating the shape of the test case's command.

B. Coverage-Directed Input Generation

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

Table I shows the targeting constraint templates for Alloy constructs based on AUnit's coverage requirements. Signature, relation and expression coverage requirements center around the size of the sets for the corresponding Alloy construct, while formula coverage requirements center around truth values. The cardinality subgroup outlines additional requirements for quantified formulas, where "d" is the domain and "b" is the formula in the body. Futher details of the coverage requirements can be found elsewhere [49], [50]. To add a targeting constraint to a model, we write the constraint as a `fact`, guaranteeing its satisfaction. We call a model with a targeting constraint a *targeting model*.

To illustrate, for our list example, let our coverage requirement of interest be "`|s| = 0`" for the signature declaration `List` (i.e., the number of List atoms equals zero). Then, the targeting constraint appended to Figure 1(a) is: `fact {#List = 0}`.

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

```
For all unexplored requirements c1 ... cn:
  c1's template || ... || cn's template
```

where 'template' refers to the unexplored coverage require-

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

TABLE I: Targeting Constraint Templates

Algorithm 2: *AGen$_{Cov}$* – Coverage-Directed Input Generation

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

ment's targeting constraint in Table I. For our running example model, the first multi-targeting constraint would cover all 35 requirements for the model: `(#List = 0) || (#List = 1) || ... || !(no n.link)`.

Algorithm 2 encapsulates our coverage-driven approach. The algorithm runs as long as there is some coverage requirement which has not yet been explored, i.e. is not covered by a previous valuation or marked infeasible. First, all coverage requirements from the model are collected into a list (*criteria*), which will store the coverage landscape of the model as the algorithm runs. Next, a targeting model is created by appending the targeting constraint, which will either reflect the next criteria to target under the unique-targeting guideline or the disjunction of all remaining criteria to explore under the multi-targeting guideline (`getNextTC`). Then, ϵ is executed over the targeting model. If the call is unsatisfiable, one of two possibilities hold: the requirements in the targeting constraint violate the facts of the model or the requirements are currently infeasible for the given scope. To determine which, we form a new targeting model with all facts removed except the targeting constraint (`removeFacts`) and execute ϵ. Should this call also be unsatisfiable, then all criteria being targeted cannot currently be covered w.r.t. the given scope and are marked infeasible.

However, should we produce a satisfiable solution space from either ϵ invocation, we have successfully targeted at least one new coverage requirement. We obtain the first instance, i, and analyze i for coverage information. We mark any requirement c that is covered for the first time as covered, removing c from the pool of requirements to target in future iterations. A requirement is known to be covered if the `evaluator` returns `true` for c's targeting constraint. Then, we generate a test case where the valuation is i and the command is either the predicate or assertion (or negation of when appropriate) that contains c. If c is located in a `fact` paragraph, the command is ϵ. For example, if Figure 1(b) is the first instance enumerated, the test ⟨Figure 1(b), run Acyclic⟩ would be created because the valuation in Figure 1(b) covers formulas and expressions in `Acyclic`, such as `#List.head = 0` and `some List.head = false`).

As with *AGen$_{BB}$*, *AGen$_{Cov}$* uses the `evaluator` to generate tests without additional invocations of SAT solvers. The `evaluator` is able to determine the shape of a test case's command when passed a requirement's origin paragraph (`c.getPara()`). Moreover, the `evaluator` is able to provide all coverage information.

IV. Mutation Testing for Alloy

This section presents μAlloy, our approach for mutation testing of Alloy models. Specifically, we present the mutation operators and the core algorithms for mutant generation, mutation score computation, equivalent mutant checking, mutation based test generation and minimal mutation test selection. Further details can be found in the second author's Masters thesis [52].

Mutation operators for Alloy Table II defines μAlloy's mutation operators, which is designed by following the convention of mutation operators for imperative languages, e.g. Java. `MOR` mutates signature multiplicity (e.g. `lone sig` to `one sig`). `QOR` mutates quantifiers `all, some, no`, etc. `UOR` and `BOR` define operator replacement for unary and binary operators. For example, `UOR` mutates `a.*b` to `a.^b` and `BOR` mutates `a=>b` to `a<=>b`. `IOBU` inserts an operator before an unary expression (e.g. `a.b` to `a.~b`). `OD` defines operator deletion (e.g. `a.* ~b` to `a.*b`). `BOE` exchanges operands for a binary operator (e.g. `a => b` to `b => a`). `IEOE` exchanges the operands of `imply-else` expression (e.g. mutate `a => b else c` to `a => c else b`). `IID` increases/decreases integer values by 1.

Algorithm 3: Mutant generation. The algorithm iterates through each Alloy Abstract Syntax Tree (AST) node and

TABLE II: Mutation Operators

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

Algorithm 3: Mutant generation algorithm

Input: Alloy model *module* and mutation operators *muOps*
Output: Valid Alloy mutants on the file system

```
// Visit each AST node and see if mutation operators apply.
while module.hasMoreASTNodes() do
    node = module.findNextASTNode()
    // Apply mutation operators to generate mutants.
    foreach muOp ∈ muOps do
        if canApplyMutationOperator(node, muOp) then
            applyAvailableMutationOperator(node, muOp)
            // If the mutant can be compiled without warning
              and error, save it.
            if isValid(module) then
                saveMutant(module)
            // Undo mutation.
            undoMutation(node)
```

Algorithm 4: Mutation score computing algorithm

Input: Alloy model *m*, mutants *mus* and AUnit test suite *ts*
Output: Mutation score

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

applies mutation operators as applicable. Once a node is mutated, μAlloy saves the new AST as a new model and compiles it. If no warning or error is found, μAlloy saves the new model to the local disk as a valid mutant.

Algorithm 4: Mutation score computation. Following traditional mutation testing, μAlloy runs the test suite against the original model and records each test outcome (as defined by AUnit). Next, μAlloy runs the test suite against each

```
one sig S {
  r: T lone->one U
}
sig T, U {}
run {}
```

(a)

```
sig S { r: T -> U }
sig T, U {}
pred sigFact() {
  one S
  all s: S | all t: T | one t.(s.r)
  all s: S | all u: U | lone (s.r).u }
run sigFact
```

(b)

Fig. 2: Signature canonicalization

Algorithm 5: Equivalent mutant checking algorithm

Input: Alloy model *m* and mutant *mu*
Output: true if equivalent, false otherwise

```
// Initialize equivalent checking model.
equivCheckModel ← emptyModel
// If mutation is in sig, cononicalize it.
if differsInSig(m, mu) then
    // canonicalizeSig() removes the constraint of the sig and
      put the constraint into a predicate. mSig and muSig are
      same; mSigConstraint and muSigConstraint are predicates.
    mSig, mSigConstraint = m.canonicalizeSig()
    muSig, muSigConstraint = mu.canonicalizeSig()
    equivCheckModel.append(mSig + mSigConstraint +
      muSigConstraint)
    // Append unaffected sig declarations.
    mSigs = extractUnaffectedSigs(m)
    equivCheckModel.append(mSigs)
    equivAssert = buildEquivCheckAssertFrom(mSigConstraint,
      muSigConstraint)
    equivCheckModel.append(equivAssert)
else
    // If mutation is not in sig, append all sigs.
    mSigs = extractAllSigs(m)
    equivCheckModel.append(mSigs)
// If mutation is in pred, add both mutated and original preds.
if differsInPred(m, mu) then
    mPred = extractPred(m)
    muPred = extractPred(mu)
    equivCheckModel.append(mPred + muPred)
    // Append unaffected pred declarations.
    mPreds = extractUnaffectedPreds(m)
    equivCheckModel.append(mPreds)
    equivAssert = buildEquivCheckAssertFrom(mPred, muPred)
    equivCheckModel.append(equivAssert)
else
    // If mutation is not in pred, append all preds.
    mPreds = extractAllPreds(m)
    equivCheckModel.append(mPreds)
// If mutation is in fact, convert the fact to a pred and add it.
if differsInFact(m, mu) then
    mFactPred = convertToPred(extractFact(m))
    muFactPred = convertToPred(extractFact(mu))
    equivCheckModel.append(mFactPred + muFactPred)
    // Append unaffected fact declarations.
    mFacts = extractUnaffectedFacts(m)
    equivCheckModel.append(mFacts)
    equivAssert = buildEquivCheckAssertFrom(mFactPred,
      muFactPred)
    equivCheckModel.append(equivAssert)
else
    // If mutation is not in fact, append all facts.
    mFacts = extractAllFacts(m)
    equivCheckModel.append(mFacts)
solutions = runEquivCheckAssertion(equivCheckModel)
// If no counter example is found, the mutant is equivalent.
if solutions.isEmpty() then
    return true
// If a counter example is found, we can use it to generate a
  test that kills the mutant. So this algorithm is same as
  mutation based test generation algorithm.
// tests = generateTests(solutions)
// saveToFiles(tests)
return false
```

Algorithm 6: Minimal tests selection greedy algorithm

Input: Alloy model *m*, mutants *mus* and test suite from mutation based test generation *ts*
Output: Approx. minimum set of tests that gives maximum mutation score

```
selectedTests ← emptySet
killedMutants ← emptySet
// For each test in the test suite, keep track of the mutants that
  can be killed by that test.
foreach t ∈ ts do
    // Get the mutant that is known to be killed by the test.
    killedMutant = t.getKilledMutant()
    // If the mutant is previously killed by a test, skip it.
    if killedMutants.contains(killedMutant) then
        continue
    // If the mutant has not been killed by any test, it is killed
      by this test so we add the test.
    selectedTests.add(t)
    // Run the test against the original model and collect result.
    mResult = runTestCase(m, t)
    // Run the test against each mutant and record the mutants
      killed by this test.
    foreach mu ∈ mus do
        // Skip mutants that are killed by the selected tests.
        if killedMutants.contains(mu) then
            continue
        // Run the test against mutant and collect result.
        muResult = runTestCase(mu, t)
        // If results differ, the mutant is killed by the test.
        if isDifferent(mResult, muResult) then
            killedMutants.add(mu)
return selecetdTests
```

mutant and collects the corrsponding test outcomes. If the test outcome for any test differs between the original model and the mutant, the mutant is killed. Finally, μAlloy computes the mutation score and reports it.

Algorithm 5: Equivalent mutant checking. We reduce the equivalent mutant checking problem to a constraint solving problem and use the Alloy tool-set to detect equivalent mutants. The reduction can lead to higher-order fomulas, so we employ Alloy*, an Alloy based toolset that provides additional support for higher-order quantifiers where allowed by the Alloy syntax [36].

To save space, we show how to handle mutations on signatures, predicates and facts. Alloy functions and assertions can be handled similar to predicates and facts. The algorithm first initializes the equivalence checking model. If the mutation happens in a signature, then we *canonicalize* the signature in both original model and mutant (Figure 2). Basically, we extract the signature constraint into a predicate and relax the signature multiplicity and relation cardinality constraints. Then, the algorithm builds an Alloy assertion to check if the signature constraints are equivalent. Finally, the algorithm adds the canonicalized signature declaration, generated predicates, all unaffected components and the equivalence checking assertion into the equivalence checking model. If no counter example is found, the original model and the mutant are equivalent – up to the chosen scope. The algorithm treats predicates and facts similarly. Specifically, to check mutation in a fact, the algorithm converts the fact into a predicate with the same body for equivalence checking (as done for signature constraints).

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

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

Algorithm 6: Mutation-based test generation. Algorithm 5 already provides a technique for test generation because the equivalence checking assertion can be used to generate counter examples when the original model and the mutant are not equivalent. The counter examples can be directly translated into tests to kill mutants. Thus, our mutation-based test generation technique first creates mutants and then creates tests that kill the non-equivalent mutants.

Algorithm 6 selects a *minimal* set of mutation based tests that kill the non-equivalent mutants. Note that each test generated by the equivalent mutant checking algorithm is known to kill at least one mutant. This algorithm uses a greedy approach and runs each test against the original model and each mutant not yet killed, and compares the test outcomes. If they differ, the mutant is killed and the algorithm records the killed mutant and selects that test.

V. Evaluation

This section presents an experimental evaluation of our test generation and mutation testing approaches. Our evaluation is two-fold. One, we use a suite of correct Alloy subjects to measure model coverage and mutation score (Section V-A). Two, we use a suite of faulty subjects with *real* faults, including standard Alloy models and graduate homework submissions, to evaluate fault finding ability (Section V-B).

A. Model coverage and mutation score

Subjects. We use seven correct Alloy models for evaluation. Four of these models, namely `SinglyLinkedList`, `BinaryTree`, `FullTree` and `N-Queens`, are written by us, and the rest are from the standard Alloy distribution. These models range from simple illustrative examples like singly-linked list to more complex examples like Dijkstra's mutual exclusion. `SinglyLinkedList`, `BinaryTree`, and `FullTree` capture the constraints of the corresponding data structures. `N-Queens` solves the problem of how to place N number of queens on a chess board of size $N \times N$ without

TABLE IV: Test suite generation and execution.

Alloy							
Model	**#Vals**	**#SAT**	**#Tests**	**T_{gen} [ms]**	**Mod Cov**	**T_{exe} [ms]**	**T_{cov}[ms]**
List	1006	1012	4010	1	89.2%	870	9
Binary Tree	1004	1010	3004	1	91.5%	1824	2
Full Tree	1004	1010	5004	1	90.5%	3410	4
Handshake	32	38	86	1	88.6%	186	15
N Queens	180	185	1888	3	84.0%	4585	56
Farmers	10	16	927	2	57.7%	205	21
Dijkstra	10	16	2893	2	64.7%	147	15
Coverage							
Model	**#Vals**	**#SAT**	**#Tests**	**T_{gen} [ms]**	**Mod Cov**	**T_{exe} [ms]**	**T_{cov}[ms]**
List	8	9	10	467	100.0%	2	2
Binary Tree	8	9	8	162	98.6%	1	2
Full Tree	8	9	15	228	95.3%	3	3
Handshake	11	12	11	423	91.4%	5	5
N Queens	7	8	13	181	97.8%	2	3
Farmers	12	13	18	1171	87.37%	25	7
Dijkstra	12	13	14	2337	87.5%	15	7
Mutation							
Model	**#Vals**	**#SAT**	**#Tests**	**T_{gen} [ms]**	**Mod Cov**	**T_{exe} [ms]**	**T_{cov}[ms]**
List	10	10	10	9	67.8%	1	1
Binary Tree	10	10	10	10	73.2%	1	1
Full Tree	15	15	15	15	65.7%	2	1
Handshake	11	11	11	13	65.8%	1	3
N Queens	17	17	17	55	73.4%	1	2
Farmers	14	14	14	29	66.4%	27	25
Dijkstra	40	40	40	123	63.0%	18	20

conflicts. `Handshake` encapsulates the Halmos handshake logic problem. `Farmers` outlines a common logic problem in which a person (the farmer) has to get three objects (fox, chicken, and grain) across a river without an object eating another. Lastly, `Dijkstra` captures Dijkstra's mutual exclusion algorithm to prevent deadlocks. Table III gives key characteristics of the subjects.

Model coverage results. Table IV shows the details for various attributes related to generation and execution of the test suites, broken down by technique. **Model** is the Alloy model under test. **#Vals** is the number of valuations generated. **#SAT** is the number of calls made to the backend SAT solver during test generation. **#Tests** is the number of test cases generated. The SAT solver is invoked either when a command in the module is executed or the constraint to create the next instance is solved. Therefore, for all three techniques, this value is the number of valuations plus the number of commands executed during test generation time. "**T_{gen} [ms]**" is the time to generate the tests, starting with the first SAT invocation and ending once writing the test(s) to a file finishes. "**Mod Cov**" shows, as a percentage, the model coverage achieved by the tests. Model coverage considers all coverage requirements produce by all Alloy elements (signatures, relations, expressions and formulas) regardless of their location within the model. Since model coverage subsumes the other coverage metrics in AUnit [50], we only use model coverage for evaluation. "**T_{exe} [ms]**" is the time to execute the tests, which is the time to check whether each valuation is an instance of its paired command. "**T_{cov}[ms]**" is the cumulative time that includes the test execution time and the time to calculate model coverage.

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

TABLE V: Mutant Generation

Module	**# gen**	**# warn**	**# error**	**# valid**	**t[ms]**
List	47	3	15	29	691
Binary Tree	158	0	93	65	1673
Full Tree	208	0	120	88	1669
Handshake	401	0	280	121	3323
N Queens	296	24	170	102	3273
Farmers	363	11	240	112	4741
Dijkstra	829	97	531	201	13598

TABLE VI: Equivalent Mutant Check with Scope of 5

Module	**# mu**	**# eq**	**# neq**	**# hi**	**t[ms]**
List	29	2	27	3	1167
Binary Tree	65	6	59	0	1216
Full Tree	88	6	82	0	1678
Handshake	121	43	78	0	2681
N Queens	102	9	93	0	3557
Farmers	112	13	99	0	5208
Dijkstra	201	14	187	3	18094

For all subjects, *AGen*$_{Cov}$ produces a relatively small test suite: no more than 18 for any subject. Test generation, execution, and coverage computation altogether take less than a seconds for most subjects. In comparison to the other techniques, the coverage-directed generation technique produces tests that give the highest coverage for all subjects. Indeed, this technique is by design constructed to focus on increasing model coverage. A nice property of the technique is that it only generates non-isomorphic tests – by construction, there are no two tests that cover the same set of requirements. Note however, coverage-directed generation does not produce all non-isomorphic tests.

AGen$_{Mu}$ produces small test suites, with the largest test suite being over the `Dijkstra` model at 40. All other models have less than 20 tests. *AGen*$_{Mu}$'s test suites are the fastest to generate, execute and calculate coverage over. *AGen*$_{BB}$'s test suites are generated faster but take longer to execute, while *AGen*$_{Cov}$'s test suites take about the same time to execute but can be slower to generate compared to *AGen*$_{Mu}$'s test suites. *AGen*$_{Mu}$ never takes coverage information into account, and usually has the lowest model coverage percentage.

Mutation Testing Results Table V shows the results for mutant generation. For each subject, the table gives the number of mutants created (**# gen**), the number of mutants that had

TABLE VII: Mutation Testing

Module	# neq	Alloy		Coverage		Mutation	
		% k	t[ms]	% k	t[ms]	% k	t[ms]
List	27	92.6	120101	85.2	664	**100**	545
Binary Tree	59	59.3	137686	54.2	894	**100**	972
Full Tree	82	70.7	429942	58.5	2426	**100**	1997
Handshake	78	76.9	341575	11.5	3363	**100**	3072
N Queens	93	87.1	2297711	69.9	4943	**100**	4099
Farmers	99	2.0	233811	2.0	2995	**100**	2986
Dijkstra	187	57.8	5039570	29.9	166154	**100**	94232

compilation warnings (**# warn**) and errors (**# error**), the resulting number of valid mutants (**# valid**) that had no warning or error, and the total generation time in milliseconds (**t[ms]**). In general, larger Alloy models have more applicable mutation operators, which leads to more mutants compared to smaller models. Complicated models also tend to have more invalid mutants. Note that some mutants of the models contain higher order logic, which the traditional Alloy analyzer cannot handle. We remove such mutants from evaluation because they can be trivially killed. The time to generate valid mutants ranges from about 7 seconds to 14 seconds, and the maximum number of mutants generated from is 201.

Table VI shows the results of the mutant equivalence checking. For each subject, the table gives the number of valid mutants (**# mu**), the number of equivalent mutants (**# eq**), the number of non-equivalent mutants (**# neq**), the number of mutants where equivalence checking introduces higher order logic (**# hi**) and the time to check equivalent mutants (**t[ms]**). Recall that while all valid mutants are themselves first order models, checking if two Alloy models are equivalent may lead to a higher order formula. We use Alloy* to solve all such higher order formulas. In our case, all our equivalent checking models can be solved using Alloy*. Among all subjects, the handshake model yields more equivalent mutants mostly because of the way it is written: it contains many constraints in the signature declarations and facts and those constraints overlap largely so that many mutations do not change the meaning of the model. The time to check equivalent mutants ranges from 1 second to 18 seconds as the number of mutants increases.

Table VII shows the results of mutation testing. For each subject, the table gives the number of non-equivalent mutants (**# neq**), the mutation score (**% k**) and the time taken for running mutation testing algorithm (**t[ms]**) with respect to test suites generated using Alloy enumeration, coverage-directed generation and mutation based minimal test generation.

The mutation based technique generates tests that kill all non-equivalent mutants, which self validates our mutation based test generation algorithm. In all subjects, the Alloy enumeration technique uniformly gives higher mutation score compared to the coverage based technique. The exception is the `farmers` model, where both techniques only kill 2.0% of the non-equivalent mutants. These poor mutation scores are because most tests generated lead to unsatisfiable formulas in test execution because the Alloy enumeration technique generates predicate invocations with all permutations of parameters. Most parameter permutations make the tests trivially unsatisfiable for the original model and all mutants, which means those tests do not kill any mutant. Another reason is that the `farmers` problem contains a very limited number of solutions given our scope of 5, which makes most tests unsatisfiable in the first place. The coverage-based technique kills only 11.5% of the mutants for the `handshake` model, it is mainly because the coverage based technique only needs to generate a small number of tests to achieve full coverage due to the nature of the model, which limits the ability to kill mutants.

B. Finding real faults

Known buggy models. We consider next two models that are faulty versions of two of our subjects in Section V-A. Faulty `Farmers` and `Dijkstra` models are from Alloy Analyzer v4.1.10 that shipped these buggy versions; the current Alloy release includes the corrected versions after the Alloy users discovered and reported the bugs. The faulty `Farmers` was *underconstrained*; the `CrossRiver` predicate checked the "eating" behavior on only one side of the river, enabling `Objects` which should have eaten each other to co-exist alone. In faulty `Dijkstra`, when the predicate `GrabbedInOrder` was invoked from `GrabOrRelease`, the model was *overconstrained* and only produced a few trivial instances. We generated test suites for each faulty model using all 3 techniques ($AGen_{BB}$, $AGen_{Cov}$ and $AGen_{Mu}$). If the test generation technique was able to produce a test which encapsulated the faulty behavior, then the bug is detected.

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

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

Table IX summarizes the results of the three techniques. The column **Fail** shows the ratio of faulty models detected with

TABLE VIII: Known Buggy Models – AUnit test generation fault finding ability.

Model	Alloy				Coverage				Mutation			
	Fail	**#Vals**	**#SAT**	**#Tests**	**Fail**	**#Vals**	**#SAT**	**#Tests**	**Fail**	**#Vals**	**#SAT**	**#Tests**
Farmers	**Y**	41	47	9769	**Y**	13	14	15	Y	12	12	12
Dijkstra	**Y**	1004	1010	287004	**Y**	17	18	61	N	36	36	36

TABLE IX: Homework submissions – AUnit test generation fault finding ability.

Alloy				Coverage				Mutation			
Fail	**#Vals**	**#SAT**	**#Tests**	**Fail**	**#Vals**	**#SAT**	**#Tests**	**Fail**	**#Vals**	**#SAT**	**#Tests**
16/19	(223,504)	(227,509)	(26296,83753)	**19/19**	(13,18)	(14,19)	(16,47)	17/19	(10,35)	(10,35)	(10,35)

respect to the total number of faulty models. All 3 technique correctly identified the non-faulty student submission model. Details of the generated test suites are shown in pairs (a,b) showing the minimum and maximum values across test suites produced over all 19 submissions. *AGen*$_{BB}$ detected bugs in a majority of the submissions, but was unable to detect a fault in 3 out of the 19 buggy models despite producing extremely large test suites. The loop-list has a number of predicates with parameters, which means a single valuation does not become a single test per command but a single test per unique combination of values for the command's parameter(s). As a result, for our evaluation, we "timed out" the technique after enumerating 500 instances, generally producing suites of size $> 80,000$. *AGen*$_{BB}$ struggled to detect faults located in `facts`, which, as expected, is a limitation of the *AGen*$_{BB}$ techniques. The solution space for negated facts is typically much larger than the valid solution space. *AGen*$_{BB}$ had greater success at finding faults in the known-buggy models, whose faults were not in the facts of the model.

AGen$_{Mu}$ had more success than *AGen*$_{BB}$ at finding faults in student submissions; however, the technique did miss faults in 2 student submissions. In an interesting case, one of the submissions that *AGen*$_{Mu}$ fails to find faults in is a submission in which the student did not fill in the predicate skeletons. In this case, *AGen*$_{Mu}$ fails to generate any tests over the predicates, thus missing the faults. Yet, even with completed predicates, faulty behavior can be missed by *AGen*$_{Mu}$, as seen by the other student submission and the faulty `Dijkstra` model.

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

Our evaluation of AUnit test generation using homework submissions made us specifically appreciate AUnit. Aside from this evaluation, we had manually graded the submissions and some of the bugs found by AUnit were previously missed by us. These bugs were usually in relation to non-obvious *compounding* formulas; Alloy's expressive nature can make it difficult to reason about a group of formulas together even if it is easy to understand them separately.

VI. Related Work

Efforts to introduce support for testing in its traditional form for Alloy can be found in a number of previous projects. For example, the Alloy tool-set has built in support for labeled commands, and allows running all commands; moreover, each command can use the `expect` clause to indicate whether a solution is expected or not. Indeed, we leverage this functionality in this paper to implement AUnit [50] tests. Moreover, Montaghami and Rayside's support for partial instances [37], [38] takes inspiration in part by the need to bring traditional testing ideas to Alloy. Furthermore, the Alloy analyzer's support for highlighting unsat cores helps with debugging faulty unsatisfiable formulas; we conjecture AUnit tests together with unsat cores provide an effective basis for introducing fault localization [26] and program repair [16] for Alloy. To our knowledge, AUnit is the only previous project that describes test cases, test execution, and model coverage (in their traditional spirit) for Alloy. This paper builds on AUnit to introduce automated test generation and mutation testing for Alloy, thereby providing a comprehensive test automation framework for Alloy.

Symmetry-breaking [15], [27], [44] substantially reduces the number of solutions to inspect and plays a key role in traditional analyses using Alloy. Aluminum [39] limits the instances to *minimal* solutions to further ease the understanding of the models using instance inspection. Our test generation techniques are orthogonal to and can be applied in synergy with symmetry breaking, minimal instances, and other scenario exploration techniques [33].

Testing constraint programs in general has been addressed in previous work. For example, a test framework was built for the constraint language OPL which focuses on using an oracle model to derive tests that look for differences in behavior based on conformity properties and provides guidance for fault localization [30], [31]. Moreover, previous work introduced a reduction of testing UML models to satisfiability checking by encoding the model and a property of interest, and using SAT [45]. Other related efforts have focused on leveraging a range of UML diagrams (sequence, class, and activity diagrams) in conjunction with pre-/post-conditions and invariants in OCL to develop a few different methodologies for guiding test input generation, stimulating execution of UML models, and observing their behavior [12], [13], [41]. Moreover, in the context of functional programs, e.g., in Haskell, automated testing is common practice [9].

Mutation testing [11], [21] is a well-studied and active research area [25]. However, in the specific context of declarative programs, mutation testing is lesser explored [6]. Previous work in this area has focused largely on mutation of specifica-

tions (typically for imperative code), where specific techniques apply mutation operators designed for imperative code but also applicable to specifications, e.g., applying relational operator replacement to replace "<" with ">". Our work on mutation testing for Alloy is closest in spirit to Srivatanakil et al. [48] who define mutation operators for CSP specifications written using FDR2 syntax [2]. Specifically, their process definition operators focus on specific specification constructs. The key difference is our support for Alloy – a relational first order logic with transitive closure. Aichernig and Salas [5] define specification mutation for OCL and apply it to pre/post-condition specifications for constraint-based testing. The mutation operators for OCL are a subset of those used commonly for imperative code, specifically for boolean expression modification. A generalization and formalization of the foundation of this work is presented in follow up work by Aichernig and Jifeng [4] who provide an integration of mutation testing with the Unifying Theory of Programming [21].

Our focus in this paper is on *testing* programs written in Alloy, a *declarative* language. There is a rich history of *using* declarative specifications to test *imperative* programs [19] with logical constraints playing an important role in systematic testing [7], [10], [17], [20], [22], [23], [29], [35], [40], [42].

While a primary role for tests is in bug finding, a number of recent projects have leveraged tests for automated debugging [16], [32], [34], [54] and program synthesis [14], [28], [46]. AUnit enables defining such approaches [53] for Alloy.

VII. Conclusion

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

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

Acknowledgements

We thank Nima Dini, Kewei Ma, Darko Marinov, and the anonymous reviewers for helpful feedback and comments. This research was partially supported by National Science Foundation Grant Nos. CCF-1319688 and CNS-1239498.

References

[1] Alloy home page. http://alloy.mit.edu.
[2] Software FDR2. http://www.fsel.com/software.html.
[3] Unified modeling language webpage. http://www.uml.org/.
[4] B. K. Aichernig and H. Jifeng. Mutation testing in utp. *Form. Asp. Comput.*, 21(1-2):33–64, Feb. 2009.
[5] B. K. Aichernig and P. A. P. Salas. Test case generation by ocl mutation and constraint solving. In *QSIC*, pages 64–71, 2005.
[6] F. Belli and O. Jack. Declarative paradigm of test coverage. *Software Testing, Verification and Reliability*, 8(1):15–47, 1998.
[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.
[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
[9] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, 2000.
[10] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, pages 215–222, Sept. 1976.
[11] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 4(11), Apr. 1978.
[12] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. A. Andrews. A tool-supported approach to testing UML design models. In *ICECCS*, pages 519–528, 2005.
[13] T. T. Dinh-trong, S. Ghosh, and R. B. France. A systematic approach to generate inputs to test UML design models. In *ISSRE*, pages 95–104, 2006.
[14] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. CodeHint: Dynamic and interactive synthesis of code snippets. In *ICSE*, 2014.
[15] J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias. TACO: Efficient SAT-based bounded verification using symmetry breaking and tight bounds. *IEEE Transactions on Software Engineering*, 39(9):1283–1307, Sept. 2013.
[16] K. Ghori. Constraint-based program repair. Master's thesis, Department of Electrical and Computer Engineering, University of Texas at Austin, Aug. 2006.
[17] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *ICSE*, pages 225–234, Cape Town, South Africa, 2010.
[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.
[19] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, pages 156–173, June 1975.
[20] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. ISSTA*, pages 53–62, Clearwater Beach, FL, 1998.
[21] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, July 1977.
[22] H.-M. Horcher. Improving software tests using Z specifications. In *Proc. 9th International Conference of Z Users, The Z Formal Specification Notation*, pages 152–166, 1995.
[23] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, 1975.
[24] D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. ICSE*, Limerick, Ireland, June 2000.
[25] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.
[26] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, May 2002.

[27] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson. A case for efficient solution enumeration. In *SAT*, May 2003.
[28] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.
[29] B. Korel. Automated test data generation for programs with procedures. In *Proc. ISSTA*, pages 209–215, San Diego, CA, 1996.
[30] N. Lazaar, A. Gotlieb, and Y. Lebbah. Fault localization in constraint programs. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 61–67, Oct 2010.
[31] N. Lazaar, A. Gotlieb, and Y. Lebbah. *Principles and Practice of Constraint Programming – CP 2010: 16th International Conference, CP 2010, St. Andrews, Scotland, September 6-10, 2010. Proceedings*, chapter On Testing Constraint Programs. 2010.
[32] F. Long and M. Rinard. Staged program repair with condition synthesis. In *ESEC/FSE 2015*, 2015.
[33] N. Macedo, A. Cunha, and T. Guimares. Exploring scenario exploration. In *FASE*, pages 301–315, 2015.
[34] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *ASE*, 2009.
[35] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. ASE*, pages 22–31, San Diego, CA, Nov. 2001.
[36] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *ICSE*, 2015.
[37] V. Montaghami and D. Rayside. Extending Alloy with partial instances. In *ABZ*, pages 122–135, 2012.
[38] V. Montaghami and D. Rayside. Staged evaluation of partial instances in a relational model finder. In *ABZ*, pages 318–323, 2014.
[39] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Aluminum: Principled scenario exploration through minimality. In *ICSE*, pages 232–241, 2013.
[40] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. Second International Conference on the Unified Modeling Language*, pages 416–429, Oct. 1999.
[41] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. France. Testing UML designs. *Information and Software Technology*, 49(8):892 – 912, 2007.
[42] C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, 1976.
[43] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.
[44] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *SAT*, June 2001.
[45] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation Test in Europe Conference Exhibition*, 2010.
[46] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.
[47] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
[48] T. Srivatanakul, J. A. Clark, S. Stepney, and F. Polack. Challenging formal specifications by mutation: A CSP security example. In *APSEC*, 2003.
[49] A. Sullivan. AUnit – A testing framework for Alloy. Master's thesis, University of Texas at Austin, May 2014.
[50] A. Sullivan, R. N. Zaeem, S. Khurshid, and D. Marinov. Towards a test automation framework for Alloy. In *SPIN*, pages 113–116, 2014.
[51] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, pages 632–647, 2007.
[52] K. Wang. muAlloy – An automated mutation system for Alloy. Master's thesis, University of Texas at Austin, May 2015.
[53] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid. Sketching Alloy models. Submitted for peer-review.
[54] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.
[55] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *ICSM*, 2010.

APPENDIX

The following partial Alloy model is to be completed with respect to the instructions given as comments in the model:

```
module list

sig List {
  header: set Node
}

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

fact CardinalityConstraints {
  // each list has at most one header node

  /* your code goes here */

  // each node has at most one link

  /* your code goes here */

  // each node has exactly one elem

  /* your code goes here */
}

pred Loop(This: List) {
  // <This> is a valid loop-list

  /* your code goes here */
}

pred Sorted(This: List) {
  // <This> has elements in sorted order ('<=')

  /* your code goes here */
}

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

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

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

  /* your code goes here */
}

abstract sig Boolean {}

one sig True, False extends Boolean {}

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

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

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