MuAlloy: A Mutation Testing Framework for Alloy

Kaiyuan Wang, Allison Sullivan, Sarfraz Khurshid
University of Texas at Austin, USA
{kaiyuanw,allisonksullivan,khurshid}@utexas.edu

Abstract

Creating models of software systems and analyzing the models helps develop more reliable systems. A well-known software modeling tool-set is embodied by the declarative language Alloy and its automatic SAT-based analyzer. Recent work introduced a novel approach to testing Alloy models to validate their correctness in the spirit of traditional software testing: AUnit defined the foundations of testing (unit tests, test execution, and model coverage) for Alloy, and MuAlloy defined mutation testing (mutation operators, mutant generation, and equivalent mutant checking) for Alloy. This tool paper describes our Java implementation of MuAlloy, which is a command-line tool that we released as an open-source project on GitHub. Our experimental results show that MuAlloy is efficient and practical. The demo video for MuAlloy can be found at https://youtu.be/3lvnQKiLcLE.

1 Introduction

Software models, which describe key properties of software systems at an abstract level, help build more reliable systems. Researchers developed various languages and tools for creating models [3, 5, 14]. A well-known software modeling tool-set is embodied by the declarative language Alloy and its automatic SAT-based analyzer [3, 12] that performs *scope-bounded* analysis with respect to a given bound on the universe of discourse.

The Alloy analyzer plays a key role in helping the users to validate their models so that they accurately reflect the intended properties. Traditionally, Alloy users employ three validation techniques: (1) solving for desired parts of the model to create *instances* that satisfy the properties modeled, e.g., acyclic structure of a network; (2) creating alternative but related (e.g., equivalent) formulations of the properties modeled, and checking whether the expected relation (e.g., equivalence) holds; and (3) using unsatisfiable cores to highlight parts of the model that cause unsatisfiability. The Alloy analyzer provides robust support for each of the three techniques.

More recent work introduced a novel approach to *testing* Alloy models in the spirit of traditional software testing so that users who are familiar with writing tests for their *imperative* code can follow a similar method for testing their *declarative* models. Specifically, the AUnit [8, 11] framework introduced the foundations of testing – including unit tests, test execution, and model coverage – for Alloy. Follow up work on MuAlloy [10, 13] introduced mutation testing [2] – including mutation operators, mutant generation, equivalent mutant checking, mutation score – for Alloy.

```
sig List { header: lone Node }
sig Node { link: lone Node }
pred Acyclic (l: List) {
  no l.header or some n: l.header.*link | no n.link  }
run Acyclic
```

Figure 1: Acyclic Singly-linked List.

This tool paper describes our Java implementation of MuAlloy, which is a command-line tool that we released as an open-source project on GitHub (https://github.com/kaiyuanw/MuAlloy). MuAlloy supports 9 mutation operators, which are inspired by previous work on mutation testing for imperative languages [4]. MuAlloy applies mutation operators at the AST level and generates a list of mutants. For each mutant, MuAlloy automatically checks if the mutant is equivalent (up to the given scope) to the original model using SAT solving. For each non-equivalent mutant found, this check creates an input that kills the mutant; MuAlloy saves both the mutant and the input (as an AUnit test) to disk, which provides mutation-based test generation. For traditional mutation testing, given an Alloy model and its test suite, MuAlloy creates non-equivalent mutants for the model and computes the mutation score for the given suite with respect to the non-equivalent mutants. We evaluated MuAlloy using 13 Alloy models that were used in previous work [6, 9, 10]. The results shows that MuAlloy is efficient and practical (it takes <10 seconds for mutant generation and <40 seconds for mutation testing for each subject model).

2 AUnit Background

Before describing the MuAlloy technique, we first describe AUnit tests. To illustrate, Figure 1 shows an acyclic singly-linked list Alloy model. The model declares a set of `"List"` and `"Node"` atoms. Each `"List"` atom has zero or one `"header"` of type `"Node"`. Each `"Node"` atom has zero or one following `"Node"` atoms named `"link"`. Both `"header"` and `"link"` are partial functions. The predicate `"Acyclic"` restricts its parameter `"l"` to be an acyclic list. The body of `"Acyclic"` states that `"l"` is acyclic if (1) it does not have a `"header"` or (2) there exists some `"Node"` reachable from `"l"`'s `"header"` following zero or more traversals of `"link"`, such that the `"Node"`'s `"link"` does not relate to any `"Node"`.

If we run the `"Acyclic"` predicate, we can get an satisfiable instance shown in Figure 2. The instance states that there are two `"List"` atoms (`"List0"` and `"List1"`) and two `"Node"` atoms (`"Node0"` and `"Node1"`). `"List0"`'s `"header"` is `"Node1"` and `"List1"`'s `"header"` is `"Node0"`. `"Node1"`'s next node is `"Node0"`. Note that `"List0"` is explicitly passed as the argument of `"Acyclic"` predicate, and we can see that `"List0"` is indeed acyclic as there is no loop in the list.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183488

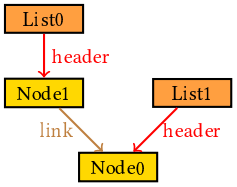

Figure 2: Satisfiable List Instance.

```
pred test {
  some disj List0, List1: List {
    some disj Node0, Node1: Node {
      List = List0 + List1
      header = List0->Node1 + List1->Node0
      Node = Node0 + Node1
      link = Node1->Node0
      Acyclic[List0] } } }
run test
```

Figure 3: List Test.

An AUnit test is a pair consisting of a model valuation and a run command. For example, the Alloy instance in Figure 2 can be written as an AUnit test as shown in Figure 3. The test declares 2 disjoint "List" atoms ("List0" and "List1") and 2 disjoint "Node" atoms ("Node0" and "Node1"). It restricts the entire "List" set to be {"List0", "List1"} and "Node" set to be {"Node0", "Node1"}. The predicate also states that the "header" maps "List0" to "Node1" and "List1" to "Node0", and the "link" maps "Node1" to "Node0". If you run the test predicate, you will obtain an isomorphic Alloy instance to the one shown in Figure 2.

3 Technique

A typical mutation testing technique has two phases:

- *Mutation (M) Phase*: Automatically inject faults into a program.
- *Testing (T) Phase*: Automatically run a given test suite against each mutated program. If some test fails, the mutant is killed.

The quality of the test suite can be gauged from the mutation score, which is the percentage of mutations killed. In this paper, the program is an Alloy model and the test suite follows the AUnit test convention.

In this section, we describe the mutation operators supported in MuAlloy. Then, we describe each phase. Finally, we describe an AST manipulation library built for MuAlloy.

Table 1: Mutation Operators

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

Algorithm 1: Mutant Generation

Input: Alloy model *M*, mutation operators *Ops*
Output: A list of non-equivalent mutants

```
Procedure visit(node, Ops, L)
    ops = findApplicable(node, Ops)
    foreach op ∈ ops do
        mutant = mutate(node, op)
        if compile(mutant) then
            if !areEquivalent(M, mutant) then
                L.add(mutant)
    foreach child ∈ node.getChildren() do
        visit(child, Ops, L)
L ← []; root = parse(M)
visit(root, Ops, L)
return L
```

A Mutation Operators

Table 1 shows the mutation operators supported in MuAlloy. *MOR* mutates signature multiplicity, e.g. "`lone sig`" to "`one sig`". *QOR* mutates quantifiers, e.g. "`all`" to "`some`", etc. *UOR*, *BOR* and *LOR* define operator replacement for unary, binary and formula list operators, respectively. For example, *UOR* mutates "`a.*b`" to "`a.^b`"; *BOR* mutates "`a=>b`" to "`a<=>b`"; and *LOR* mutates "`a&&b`" to "`a||b`". *UOI* inserts an unary operator before an expression, e.g. "`a.b`" to "`a.~b`". *UOD* deletes an unary operator from an expression, 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. "`a => b else c`" to "`a => c else b`". All mutation operators are defined at the AST level and modifying AST nodes properly is non-trivial. For example, "`&&`" and "`||`" are list operators in Alloy. Replacing "`&&`" with "`||`" in "`a||(b&&(c||d))`" should results in "`a||b||c||d`", which means we need to properly flatten the parent and child AST nodes after mutation.

B Mutation (M) Phase

MuAlloy applies mutation operators to Alloy AST nodes. Algorithm 1 describes how MuAlloy generates mutants from a given model. The algorithm takes as input an Alloy model *M* and the set of predefined mutation operators in Table 1. The output is a list of non-equivalent mutant models *L*. MuAlloy first initializes the result list *L* as empty. Then, MuAlloy parses the target Alloy model as an AST and returns the root node. MuAlloy implements a visitor pattern to visit each AST node recursively using a depth first search. For each visited node, MuAlloy finds all the applicable mutation operators and applies each operator to the node one at a time. The *mutate* method modifies the AST *node* and returns a clone version of the mutated model (*mutant*). The *mutant* is added to the result list *L* only if it compiles and is not equivalent to the original model *M*. The *areEquivalent* method checks if the mutated model is equivalent to the original model or not using Alloy's built-in "`check`" command. For each mutation, MuAlloy only checks equivalence of the affected construct declared in *M*, which avoids redundant checks and saves time. For example, if MuAlloy mutates "`sig List { header: lone Node }`" to "`sig List { header: one Node }`", then

Algorithm 2: Mutation Testing

Input: Alloy model *M*, generated mutants *U*, test suite *T*
Output: The mutation score for test suite *T*

```
killed = 0; resM = []
foreach test ∈ T do resM.add(run(test, M))
foreach mutant ∈ U do
    foreach i ∈ 1..T.size() do
        if resM[i] != run(T[i], mutant) then
            killed++
            break
return killed / U.size()
```

the only affected Alloy construct is the `"List"` signature declaration. MuAlloy canonicalizes the signature declaration as `"sig List { header: set Node }"` and automatically generates an equivalence checking command as `"check { all l: List | lone l.header <=> one l.header }"`. Then, MuAlloy invokes the Alloy Analyzer to run the above `"check"` command. If no counterexample is found, the mutant is equivalent to the original model and MuAlloy will not add the mutant to *L*. If a counterexample is found, then MuAlloy optionally encodes the counterexample into an AUnit test which kills the mutant. In the end of the this phase, MuAlloy generates a test suite that kills all mutants and returns all non-equivalent mutants (*L*).

C Testing (T) Phase

MuAlloy can compute the mutation score for a given AUnit test suite. Algorithm 2 describes how MuAlloy runs mutation testing. The algorithm takes as input an Alloy model *M*, a set of mutant models *U* and a given test suite *T*. The output is the mutation score for the test suite *T*. MuAlloy initiates the number of killed mutants (*killed*) to 0. Then, MuAlloy runs each AUnit test in *T* under *M* and collects test results for the entire test suite as *resM*. The *run* method invokes an AUnit test under a given model and returns a boolean result indicating whether the test is satisfiable or not. For each mutant model, MuAlloy runs each test and checks if the test result varies from that of the original model *M*. If the test result is different, the mutant model is marked as killed by increasing *killed* variable and the algorithm checks the next mutant. If all test results for the mutant model are the same as those of the original model, then the test suite *T* does not kill the mutant. Finally, the algorithm returns the mutation score indicating the percentage of mutants killed along with the total number of mutants.

In practice, a user can provide an AUnit test with the `"expect"` keyword, which indicates the expected satisfiability of the test. With the expected satisfiability of all tests, *M* is not needed in Algorithm 2. However, since the expected test output should be examined manually, we decide to not generate AUnit tests in the mutation phase with `"expect"` keywords. We made this decision because a given Alloy model may be faulty and assuming the ground truth based on a potentially faulty model does not make sense.

D AST Traversal Library

To make MuAlloy robust and easy to extend, we implemented a stand-alone Alloy parser library, i.e. AlloyParser. AlloyParser follows a similar design methodology to the JavaParser [1] and includes a set of visitors to help users traverse and make updates to any Alloy AST. Although the Alloy tool-set comes with visitor classes like `"VisitQuery"` and `"VisitReturn"`, they only support visiting an Alloy expression. Our visitors are much more powerful as they allow users to visit arbitrary nodes in the AST. AlloyParser also extends existing Alloy AST nodes with richer types. The standard Alloy comes with a very compact AST node representation and does not distinguish expressions from formulas in the implementation. This distinction may be useful in some cases. For example, a user may want to define different mutation operators for expressions and formulas, and with richer AST nodes, e.g. if formulas and expressions are represented by different types of AST nodes, the user does not need to repeat the logic that checks if an AST node represents a formula or an expression. This check may be duplicated many times because many AST nodes in the original Alloy tool-set represent both expressions and formulas, e.g. `"ExprUnary"`, `"ExprBinary"` and `"ExprQt"`, etc. AlloyParser also comes with a `"PrettyStringVisitor"` class and a `"CloneVisitor"` class. The `"PrettyStringVisitor"` class converts an Alloy AST node and its subnodes to code fragments. The `"CloneVisitor"` class creates a deep copy of an Alloy AST node and its subnodes. MuAlloy's mutation phase is mainly built on top of the AlloyParser by extending and using the AlloyParser's visitors.

4 Usage

In this section, we describe how users can run MuAlloy. MuAlloy is a command line tool that comes with the following features:

- Generate non-equivalent mutants and mutant killing AUnit tests.
- Run mutation testing for a given test suite.

In this section, we discuss how to use these features. More details can be found on the MuAlloy GitHub homepage.

A Generate Mutants and Tests

To generate non-equivalent mutants and mutant killing tests, run `"./mualloy.sh --generate-mutants -o <arg> -m <arg> [-s <arg>] [-t <arg>]"` or `"./mualloy.sh --generate-mutants --model-path <arg> --mutant-dir <arg> [--scope <arg>] [--test-path <arg>]"`. The options are explained below:

- `"-o,--model-path"`: This argument is *required*. Pass the model you want to mutate as the argument.
- `"-m,--mutant-dir"`: This argument is *required*. Pass the directory to which you want to save mutants as the argument. If the directory does not exist, a new directory will be created.
- `"-s,--scope"`: This argument is *optional*. Pass the Alloy scope for equivalence checking. For each non-equivalent mutant, the scope will also be used to generate a run command for the corresponding AUnit test that kills the mutant. If the argument is not specified, a default value of 3 is used.
- `"-t,--test-path"`: This argument is *optional*. Pass the path to which you want to save mutant killing test suite as the argument. If the argument is not specified, no mutant killing test suite will be generated. Note that the generated test suite only contains unique test predicates and the run commands.

For each model, the command reports the number of equivalent mutants, non-equivalent mutants and unique AUnit tests generated by MuAlloy.

B Run Mutation Testing

For mutation testing, run "`./mualloy.sh --run-mutation-testing -o <arg> -m <arg> -t <arg>`" or "`./mualloy.sh --run-mutation-testing --model-path <arg> --mutant-dir <arg> --test-path <arg>`". The options are explained below:

- "`-o,--model-path`": This argument is *required*. Pass the original model as the argument. MuAlloy collects the test satisfiability result for the original model and then compares it with the test result for a mutant model. If the results are different, then the mutant is killed.
- "`-m,--mutant-dir`": This argument is *required*. Pass the directory to which mutants are saved as the argument. MuAlloy collects test results for each of the mutant models and checks if it can be killed by the test suite or not.
- "`-t,--test-path`": This argument is *required*. Pass the test suite you want to run as the argument. MuAlloy runs the test suite against the original model and mutant models to compute the mutation score for the test suite. Note that the test suite should only contain the test predicates and the run commands.

The command reports whether each individual mutant is killed by the test suite or not. After MuAlloy finishes running the test suite against all mutants, the command reports the mutation score.

5 Evaluation

This section describes the experiment setup and results for MuAlloy. We ran MuAlloy on a MacBook Pro with a 2.5 GHz Intel Core i7-4870HQ. Table 2 shows the 13 Alloy models involved in the experiment and the results when running MuAlloy. **Model** shows the model names. Address book (**addr**) and Grandpa (**grand**) are from Alloy's example set. Grade book (**grade**), bad employee (**bempl**) and other groups (**other**) are Alloy translations of access-control specifications [7]. **btree** models binary trees. **ctree** models two colored undirected trees. **dijkstra** models how mutexes are grabbed and released by processes, and how Dijkstra's mutex ordering criterion can prevent deadlocks. **farmer** models the farmer-crossing-river problem. **fullTree** models full binary trees. **hshake** models the Halmos handshake problem. **nqueens** models the N queens problem. **list** models acyclic singly-linked lists. **#ast** shows the number of AST nodes in each model. **scp** shows the scope used to check equivalence and generate tests. **#eq** and **#neq** show the number of equivalent and non-equivalent mutants for each model, respectively. **#test** shows the number of unique tests created by MuAlloy. $\mathbf{T}_{gen}$ shows the mutant generation time. $\mathbf{T}_{test}$ shows the mutation testing time.

Table 2: MuAlloy Subject Stats. Times are in seconds.

Model	#ast	scp	#eq	#neq	#test	T_{gen}	T_{test}
addr	114	4	4	58	43	1.9	4.3
bempl	46	3	1	30	25	1.0	2.2
btree	58	3	11	67	24	1.7	2.2
ctree	71	3	19	78	22	2.3	3.3
dijkstra	385	3	13	145	83	8.2	39.0
farmer	169	4	10	93	48	4.5	10.1
fullTree	81	3	17	83	28	2.9	3.6
grade	64	3	2	34	28	1.3	3.3
grand	96	4	13	87	40	3.0	8.7
hshake	127	5	43	92	30	3.8	6.7
nqueens	104	4	10	54	33	2.9	8.2
other	64	3	3	34	20	1.3	2.4
list	35	3	5	28	17	1.2	1.8

The most complex subject is **dijkstra** (385 nodes) and the simplest subject is **list** (35 nodes). The number of equivalent mutants is relatively small compared to the number of non-equivalent mutants. The number of unique tests is strictly smaller than the the number of non-equivalent mutants because MuAlloy removes duplicate AUnit test cases that kill multiple mutants. The time to generate mutants ranges from 1.0 to 8.2 seconds while the time to run mutation testing for all mutants given a model ranges from 1.8 to 39.0 seconds. In general, MuAlloy generates more mutants and tests for more complex models and takes a longer time to generate mutants and run mutation testing in these cases. Overall, these results show that both mutant generation and mutation testing are fast and practical using MuAlloy.

6 Conclusion

This paper introduced the MuAlloy tool for mutation testing of Alloy models. MuAlloy provides command line options to automatically generate non-equivalent mutants as well as tests that kill non-equivalent mutants. Given an Alloy model and its test suite, MuAlloy reports the mutation score for the test suite against non-equivalent mutants. MuAlloy additionally provides an Alloy AST manipulation library. Our evaluation shows that MuAlloy is efficient and practical. Moreover, MuAlloy provides a sound basis for developing new techniques and tools that leverage mutation testing, e.g., for fault localization.

Acknowledgments

We would like to thank the anonymous reviewers and Hayes Converse for the valuable comments and helpful suggestions. The work is partially supported by the US National Science Foundation under Grant No. NSF CCF-1718903.

References

[1] 2017. JavaParser. (2017). http://javaparser.org
[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* (1978).
[3] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM TOSEM* (2002).
[4] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE TSE* (2011).
[5] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *ECOOP*.
[6] Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. 2017. The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance. In *ESEC/FSE*.
[7] Salman Saghafi, Ryan Danas, and Daniel J. Dougherty. 2015. Exploring Theories with a Model-Finding Assistant. In *CADE*.
[8] Allison Sullivan. 2014. *AUnit - A Testing Framework for Alloy*. Master's thesis. University of Texas at Austin.
[9] Allison Sullivan, Kaiyuan Wang, Sarfraz Khurshid, and Darko Marinov. 2017. Evaluating State Modeling Techniques in Alloy. In *SQAMIA*.
[10] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *ICST*.
[11] Allison Sullivan, Razieh Nokhbeh Zaeem, Sarfraz Khurshid, and Darko Marinov. 2014. Towards a Test Automation Framework for Alloy. In *SPIN*.
[12] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS*.
[13] Kaiyuan Wang. 2015. *muAlloy – An Automated Mutation System for Alloy*. Master's thesis. University of Texas at Austin.
[14] Jos Warmer and Anneke Kleppe. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*.