

Copyright

by

Jiaolong Yu

2017

**Report Committee for Jiaolong Yu**  
**Certifies that this is the approved version of the following report:**

**A Prototype Implementation of the AUnit Test Automation Framework  
for Alloy**

**APPROVED BY**  
**SUPERVISING COMMITTEE:**

---

Sarfraz Khurshid, Supervisor

---

Razieh Nokhbeh Zaeem

**A Prototype Implementation of the AUnit Test Automation Framework  
for Alloy**

**by**

**Jiaolong Yu, B.E.**

**Report**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Science in Engineering**

**The University of Texas at Austin**

**May 2017**

## **Acknowledgements**

I would like to thank Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem and Sarfraz Khurshid for their continuous help and guidance through my implementation of the test automation and my master report.

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-1319688 and CNS-1239498).

## **Abstract**

# **A Prototype Implementation of the AUnit Test Automation Framework for Alloy**

Jiaolong Yu, MSE

The University of Texas at Austin, 2017

Supervisor: Sarfraz Khurshid

Alloy is a declarative language based on relational first-order logic. Unlike commonly used procedural languages, the testing criteria of declarative languages like Alloy has remained largely ad hoc. Recent work on the AUnit test automation framework introduced a foundation for testing Alloy models. This report presents our effort on developing a prototype implementation of AUnit based on the standard Alloy distribution. Our implementation of AUnit has all core functionalities for writing unit tests, running all tests, showing the test execution results including the number of tests ran, the number of tests failed, coverage obtained (which is highlighted using coloring), all test requirements, and all uncovered requirements. We compute coverage for signatures, fields, predicates and specifically for primitive Booleans and quantified formulas. Our implementation can allow users to check the quality of their models in the spirit of traditional unit testing.

## Table of Contents

List of Tables .....	viii
List of Figures .....	ix
Chapter 1 Introduction .....	1
Alloy .....	1
Alloy Analyzer .....	2
Motivation for Alloy Test Automation .....	2
Chapter 2 Background: AUnit coverage criteria .....	3
AUnit coverage criteria .....	3
New thoughts for criteria used in Implementation .....	5
Chapter 3 Illustrative example .....	7
User Interface .....	7
Example 1 .....	8
Example 2 .....	10
Example 3 .....	12
Chapter 4 Implementation.....	13
Basic Settings.....	13
Running the Tests .....	13
Evaluation and Coverage .....	14
General Implementation.....	15
Tree for Quantified Formula Evaluation.....	16
Chapter 5 Discussion and Future Work .....	22
Coverage for Tests with No Instance .....	22
A Better Data Structure.....	22
Chapter 6 Conclusion.....	24
Appendix: Code of QtTestTree.....	25

## **List of Tables**

Table 2.1: Requirements for different entities .....	4
------------------------------------------------------	---

## List of Figures

Figure 2.1: Basic structure of an Alloy Module .....	4
Figure 3.1: Setting test label prefix .....	7
Figure 3.2: Running all tests .....	8
Figure 3.3: Code and results for example 1 with one test.....	9
Figure 3.4: Code and results for example 1 with three tests .....	10
Figure 3.5: Code and partial result for example 2 .....	10
Figure 3.6: All Test Requirements of signatures, fields and predicates for example 2 .....	11
Figure 3.7: All requirements of formulas inside predicates for example 2 .....	11
Figure 3.8: Code and results for example 3 .....	13
Figure 4.1: Invoking doRun(-6) .....	14
Figure 4.2: Method “run” in SimpleTask1 .....	14



## **Chapter 1: Introduction**

This report focuses on developing an prototype implementation of the test automation framework AUnit [1,5] for the Alloy language [4] integrated with the Alloy Analyzer. It also introduces some additional thoughts into the testing criteria to achieve a reasonable implementation. It gives an option to run all test cases and show the result of testing, including the number of test cases run, the number of test cases failed, all requirements for the input, all requirements not covered and the coverage of the test cases. As for the implementation of coverage, it follows the convention of Emma for Java. It also visualizes the coverage status by coloring the input.

This chapter explains Alloy and Alloy Analyzer and motivates Alloy test automation. Chapter 2 explains in detail the testing criteria foundation [1] and additional practical ideas used in the implementation. Chapter 3 gives some illustrative examples of this test automation with explanations and pictures of user interface. Chapter 4 illustrates the implementation of the test automation including the data structure used and its code snapshots. Chapter 5 discusses the achievements and limitations of this implementation and future work to make this test automation more robust. Chapter 5 concludes this report.

### **ALLOY**

Alloy is a declarative language. Instead of giving the control flow of the computation, it specifies the conditions and logic of it. More specifically, it is a language based on first order logic that simulates models structurally. It specifies basic elements and the relationship between them as well as constrains that need to hold for the entire model. First developed by the Software Design Group at MIT in 1997, it was designed to serve as a “model finder”. With all elements and constrains specified, it finds an instance

for the whole model that has a set of atoms for each element and relationship and all the constraints hold within this instance. It is widely used in all kinds of system modeling with an emphasis on the ones that involve complex structured state. Applications of Alloy include: name servers, network configuration protocols, access control, telephony, scheduling, document structuring, key management, cryptography, instant messaging, railway switching, filesystem synchronization, and semantic web.[2]

### **ALLOY ANALYZER**

The Alloy Analyzer is essentially a compiler of Alloy language. It uses SAT solvers to find a valid instance for a user-specified Alloy model or show that there is none. It regard the whole model as a huge Boolean formula and hand it over to a SAT solver. It retrieves the result and translates it into the language of model. It provides different SAT solvers for users to choose from and also provides an evaluator for every instance found in case of further exploration. It provides detailed output and error report for the user input Alloy file, as well as an abstract syntax tree that gives a clear structure of the model.

### **MOTIVATION FOR ALLOY TEST AUTOMATION**

As stated previously, Alloy is used to design and analyze complex software models. However, it is usually difficult for people to model a complex model by hand. It is very common to have small defects inside a large model. Thus it is very important to build a test automation for Alloy users. With the test automation, users will be able to check whether their model is logically comprehensive. For that purpose, a recent work [1] came up with a theory prototype of a test automation for Alloy.

## Chapter 2: Background: AUnit coverage criteria

This chapter gives a brief introduction to the important background of this report, AUnit coverage criteria. It also introduces some further details to have a reasonable implementation.

### AUNIT COVERAGE CRITERIA

The idea of AUnit was first introduced by Sullivan et al. [1] Here we briefly illustrate the key coverage criteria for AUnit.

An Alloy module is a virtual logic system built by the Alloy language with user defined constrains. In a module, there exists a list of signatures, a list of facts, a list of assertions and a list of predicates.

- Signatures are basic elements in the module, which are like classes in Object Oriented Programming. Instead of instantiating them by programs, every instance found by Alloy Analyzer will give a set of instances for each signature. That set can have 0, 1 or more than 1 instance in it. In every signature there can be a list of fields, which represents a projection from one signature to another. A field can also be viewed as a relationship that exists from one signature to another. Similarly every instance gives a set of instances of each field.
- Facts and assertions specify constrains that apply to the signatures and fields. Facts are assumptions of the model. Assertions are intended to be followed.
- Predicates are similar to functions whose return type is Boolean.

Every fact, assertion or predicate, has a body that is a tree of expressions and formulas. Every expression or formula can be regarded as a node with a possible list of

sub-nodes. The basic structure is given by Fig 2.1. The module also has a list of commands. A command is an instruction to analyze the module. It is either a run or a check. The model regards the body of run as a predicate, which similarly to facts, has to be followed. The body of check is regarded as an assertion.

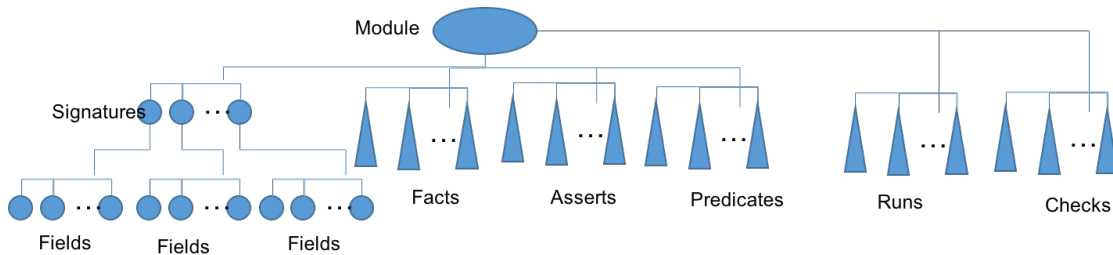


Fig 2.1 Basic structure of an Alloy Module

Sullivan et al. introduce the following table of test requirements for all entities in the module.

Entities	Requirements	
Signature $s$	1. $ s  = 0$ ; 2. $ s  = 1$ ; 3. $ s  \geq 2$	
Field $f$	1. $ f  = 0$ ; 2. $ f  = 1$ ; 3. $ f  \geq 2$	
Expression $e$ (evaluates to a set)	1. $ e  = 0$ ; 2. $ e  = 1$ ; 3. $ e  \geq 2$ .	
Formula $fm$ (evaluates to a Boolean)	1. $!fm$ ; 2. $fm$	
Quantified Formula $qt$ say " $Q x : d \mid b$ " with quantifier $Q$ , variable $x$ , domain $d$ , and body $b$	$ d  = 0$	
	$ d  = 1$	$b$ is True
		$b$ is False
	$ d  \geq 2$	$b$ is True for each atom in $d$
$b$ is False for each atom in $d$		
$b$ is true for at least one atom in $d$ , and is false for at least one atom in $d$ .		

Table 2.1 Requirements for different entities

## NEW THOUGHTS FOR CRITERIA USED IN IMPLEMENTATION

The criterion illustrated in the AUnit paper [1] is thorough. However, the expressions and formulas can be nested inside other expressions and formulas. According to the coverage criteria introduced in *Introduction in Software Testing* [3], there are only two well defined coverage for predicates: predicate coverage and clause coverage. It is redundant to look at coverage at every node (expression or formula), because usually the values of intermediate nodes are not essential to the module. So in our implementation, a notion similar to clause coverage is used. The implementation specifically looks at primitive Boolean formulas to compute coverage. Since predicates, assertions and facts have their own coverage requirements, predicate coverage is also included. For example, consider this predicate:  $p\{\text{some } a \text{ and } (\text{some } b \text{ and } \text{lone } c)\}$ . We have coverage requirements for  $p$  to be true or false. We also have coverage requirements for “some  $a$ ”, “some  $b$ ” and “some  $c$ ”, however, we do not have coverage requirements for “some  $b$  and lone  $c$ ”. In the original criteria defined in the AUnit paper [1], we would also have requirements for “some  $b$  and lone  $c$ ” since it is also a formula.

In addition, we calculate coverage for every quantified formula. It means we calculate coverage for all nodes that are quantified formulas. For quantified formulas with multiple declarations, for example: “ $Q x,y:d1, z:d2 \mid b$ ”, our implementation reconstruct the formula to nested quantified formulas and calculates coverage for each of the quantified formulas. For example, “ $Q x,y:d1, z:d2 \mid b$ ” becomes “ $Q x:d1 \mid Q y:d1 \mid Q z:d2 \mid b$ ” thus yeilds 18 requirements in total for three quantified formulas. Variables  $x$ ,  $y$  and  $z$  are used in body  $b$ . More generally for nested quantified formula, variable  $x$  can be used in “ $Q y:d1 \mid Q z:d2 \mid b$ ”, since it is the body of this formula. At every quantified formula, every possible value for variables are used or tried out in its body to calculate the coverage for the body, which may be a simple formula or a nested quantified formula.

Another important concept to clarify is the definition of a pass or failed test. A test passes when the analyzer finds an instance for the test when the test expects one or the analyzer finds no instance for the test when the expects none. All other circumstances are regarded as failed. Only tests with an instance contribute to coverage.

## Chapter 3: Illustrative example

### USER INTERFACE

As mentioned in the introduction, our implementation is integrated with Alloy Analyzer. It is designed so that all commands labeled with some specific prefix are regarded as tests. Users are given the option to set the prefix as they wish. This functionality is put under the menu of “Options” shown below in Fig 2.1.

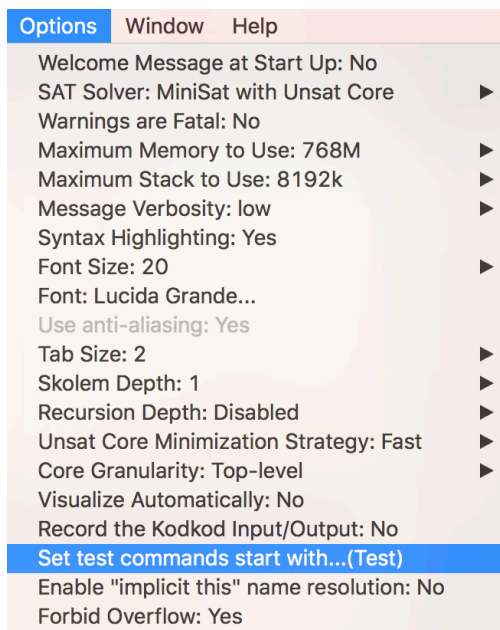


Fig 3.1 Setting test label prefix

Clicking the button “Execute All Tests” in the execute menu will run all the tests and output the results in the console on the right side. As shown in Fig 3.2 and Fig 3.3.

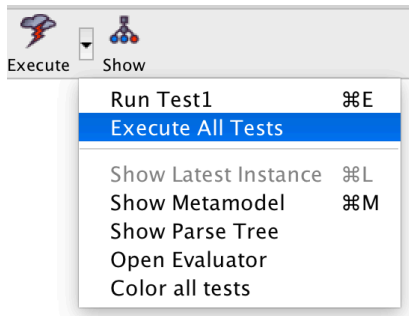


Fig 3.2 Running all tests

### EXAMPLE 1

This is a very simple example to show the core functionality of the test automation. The input Alloy code and the test result is shown below in Fig 3.3. The window named “Run Test1” is opened by clicking the “Instance” in the console on the right side. It shows the first instance found by the analyzer that satisfies the constrain given by the declaration of signature and field and the command of Test1. In the result, except for the output analyzer used for running a command, the information of pass and fail, the coverage status, the test requirements and uncovered requirements are all given. By clicking the “Coverage:” in the result, the coloring of coverage status is shown in the input window.

One can see that there is only one Signature and one Field in the code. Thus according to the testing criterion, we only have 6 requirements:

- 1) #this/S = 0
- 2) #this/S = 1
- 3) #this/S >= 2
- 4) #f = 0
- 5) #f = 1
- 6) #f >= 2



And since we set the test prefix as “Test”, only the first command is supposed to be run as a test. According to the result, the program behaved as expected and the test requirements are also correctly displayed. That instance is what we use to find out about the coverage of this test. As we can see in the instance, it has one S and one f. Thus requirements 2) and 5) are satisfied. Therefore, the rest of the test requirements are uncovered, which is shown in the “Uncovered Test Requirements” section of output. We covered 2 requirements out of 6, thus the coverage is 33%. And both S and f are partially covered thus they are colored with orange in the input area.

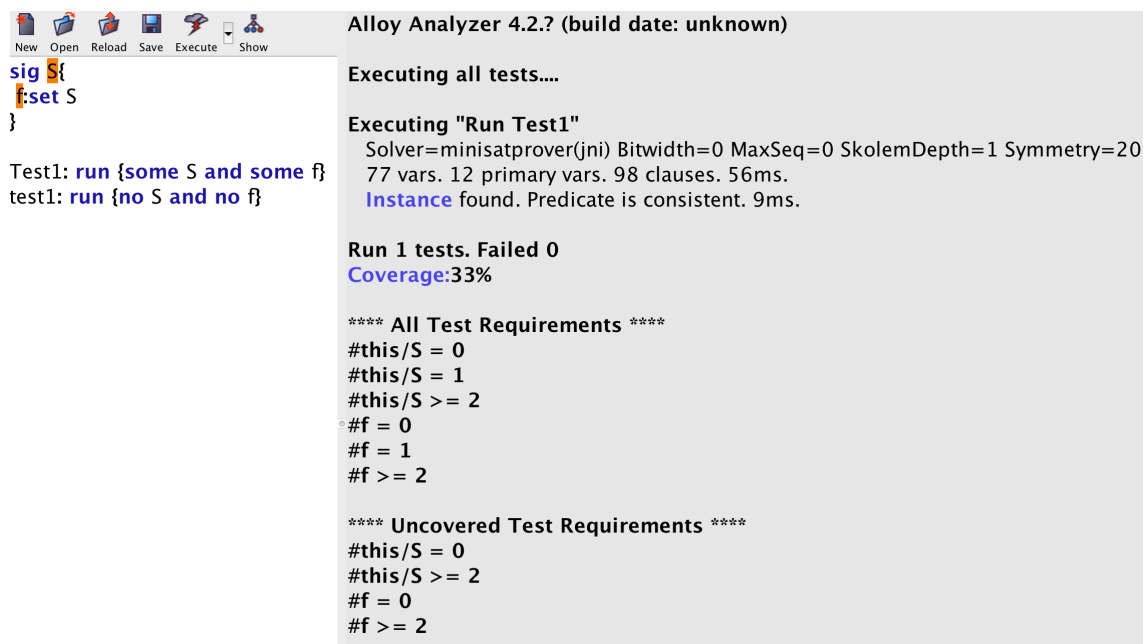


Fig 3.3 Code and results for example 1 with one test

If we have more tests, like in Fig 3.4, we have more coverage and S is fully covered. In that case, S is shaded with green. If an entity, like signature or predicate, is not covered at all, it is colored in red. The case of no coverage will be shown in example 3.

```

New Open Reload Save Execute Show
sig S {
  f: set S
}

Test1: run {some S and some f}
Test2: run {no S and no f}
Test3: run{some S and all s: S| some (S - s)}

Executing "Run Test2"
Solver=minisatprover(jni) Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
77 vars. 12 primary vars. 108 clauses. 4ms.
Instance found. Predicate is consistent. 3ms.

Executing "Run Test3"
Solver=minisatprover(jni) Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
83 vars. 12 primary vars. 106 clauses. 5ms.
Instance found. Predicate is consistent. 2ms.

3 commands were executed. The results are:
#1: Instance found. Test1 is consistent.
#2: Instance found. Test2 is consistent.
#3: Instance found. Test3 is consistent.

Run 3 tests. Failed 0
Coverage:83%

**** All Test Requirements ****
#this/S = 0
#this/S = 1
#this/S >= 2
#f = 0
#f = 1
#f >= 2

**** Uncovered Test Requirements ****
#f >= 2

```

Fig 3.4 Code and results for example 1 with three tests

## EXAMPLE 2

In this example we give a more complex model that includes crucial expressions of Alloy grammar like predicates and quantified formulas, as shown below in Fig 3.5

```

New Open Reload Save Execute Show
Untitled 1 - dijkstra
sig S {
  f: set T
}

sig T {
  g: set R
}

sig R {
  h: set S
}

pred p {
  some S and all t1, t2: S.f| all r:t1.h|some t2 and some r
}

Test1: run {p}
Test2: run {!p}

Run 2 tests. Failed 0
Coverage:45%

**** All Test Requirements ****
#this/S = 0
#this/S = 1
#this/S >= 2
#this/T = 0
#this/T = 1
#this/T >= 2
#this/R = 0
#this/R = 1
#this/R >= 2
#f = 0
#f = 1
#f >= 2
#g = 0
#g = 1
#g >= 2
#h = 0

```

Fig 3.5 Code and partial result for example 2

Apart from what we had in Example 1, we also have coverage for predicate p in this example. The test requirements for signatures, fields and predicates are shown in Fig

3.6. The requirements inside the predicate p are shown in Fig 3.7. As we can see here we have 2 requirements, true and false for predicates and also primitive Boolean inside predicates. Also we have 6 requirements for each of quantified formula, including nested ones.

```
**** All Test Requirements ****
#this/S = 0
#this/S = 1
#this/S >= 2
#this/T = 0
#this/T = 1
#this/T >= 2
#this/R = 0
#this/R = 1
#this/R >= 2
#f = 0
#f = 1
#f >= 2
#g = 0
#g = 1
#g >= 2
#h = 0
#h = 1
#h >= 2
this/p = true
this/p = false
```

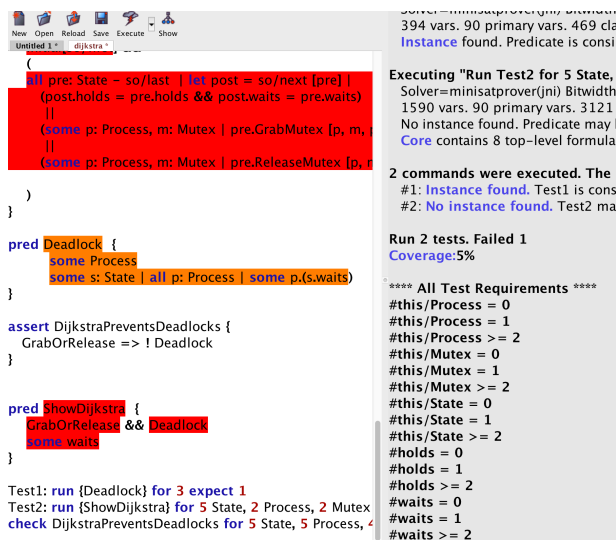
Fig 3.6 All Test Requirements of signatures, fields and predicates for example 2

```
some this/S = true
some this/S = false
#( this/S ) . ( (this/S <: f) ) = 0
#( this/S ) . ( (this/S <: f) ) = 1 and (all t2:( this/S ) . ( (this/S <: f) ) | (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] = true) ) = true
#( this/S ) . ( (this/S <: f) ) = 1 and (all t2:( this/S ) . ( (this/S <: f) ) | (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] = false) ) = true
#( this/S ) . ( (this/S <: f) ) >= 2 and (all t2:( this/S ) . ( (this/S <: f) ) | (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] = true for all var in the domain) ) = true
#( this/S ) . ( (this/S <: f) ) >= 2 and (all t2:( this/S ) . ( (this/S <: f) ) | (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] = false for all var in the domain) ) = true
#( this/S ) . ( (this/S <: f) ) >= 2 and (all t2:( this/S ) . ( (this/S <: f) ) | (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] is false for at least one var in the domain) ) = true
and also is true for at least one var in the domain
-----In this Qt-----
#( this/S ) . ( (this/S <: f) ) = 0
#( this/S ) . ( (this/S <: f) ) = 1 and (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] = true) = true
#( this/S ) . ( (this/S <: f) ) = 1 and (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] = false) = true
#( this/S ) . ( (this/S <: f) ) >= 2 and (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] = true for all var in the domain) = true
#( this/S ) . ( (this/S <: f) ) >= 2 and (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] = false for all var in the domain) = true
#( this/S ) . ( (this/S <: f) ) >= 2 and (all r:( t1 ) . ( (this/R <: h) ) | AND[some t2, some r] is false for at least one var in the domain) = true
and also is true for at least one var in the domain
-----In this Qt-----
#( t1 ) . ( (this/R <: h) ) = 0
#( t1 ) . ( (this/R <: h) ) = 1 and AND[some t2, some r] = true
#( t1 ) . ( (this/R <: h) ) = 1 and AND[some t2, some r] = false
#( t1 ) . ( (this/R <: h) ) >= 2 and AND[some t2, some r] = true for all var in the domain
#( t1 ) . ( (this/R <: h) ) >= 2 and AND[some t2, some r] = false for all var in the domain
#( t1 ) . ( (this/R <: h) ) >= 2 and AND[some t2, some r] is false for at least one var in the domain
and also is true for at least one var in the domain
-----In this Qt-----
some t2 = true
some t2 = false
some r = true
some r = false
```

Fig 3.7 All requirements of formulas inside predicates for example 2

### EXAMPLE 3

This is an example for a fairly complicated input to show the capabilities of this test automation. The input used is an example model in the Alloy Analyzer. It is an algorithm called Dijkstra. We changed the first two commands into tests. In this case since there are only two tests and one failed, the coverage is very low and the input is mostly colored by red. The failed test does not contribute to the coverage, because there is no instance to consider.



```

New Open Reload Save Execute Show
dijkstra

pre: State ~ so/last | ! post = so/next [pre] |
(post.holds = pre.holds && post.waits = pre.waits)
||
(some p: Process, m: Mutex | pre.GrabMutex [p, m,
||
(some p: Process, m: Mutex | pre.ReleaseMutex [p,
)
}
}

pred Deadlock {
some Process
some s: State | all p: Process | some p.(s.waits)
}

assert DijkstraPreventsDeadlocks {
GrabOrRelease => ! Deadlock
}

pred ShowDijkstra {
GrabOrRelease && Deadlock
some waits
}

Test1: run {Deadlock} for 3 expect 1
Test2: run {ShowDijkstra} for 5 State, 2 Process, 2 Mutex
check DijkstraPreventsDeadlocks for 5 State, 5 Process, 4

394 vars, 90 primary vars, 469 cla
Instance found. Predicate is consis

Executing "Run Test2 for 5 State,
Solver=minisatprover(jni) Bitwidth:
1590 vars, 90 primary vars, 3121
No instance found. Predicate may b
Core contains 8 top-level formula

2 commands were executed. The
#1: Instance found. Test1 is consi
#2: No instance found. Test2 ma

Run 2 tests. Failed 1
Coverage:5%

**** All Test Requirements ****
#this/Process = 0
#this/Process = 1
#this/Process >= 2
#this/Mutex = 0
#this/Mutex = 1
#this/Mutex >= 2
#this/State = 0
#this/State = 1
#this/State >= 2
#holds = 0
#holds = 1
#holds >= 2
#waits = 0
#waits = 1
#waits >= 2

```

Fig 3.8 Code and results for example 3

## Chapter 4: Implementation

### BASIC SETTINGS

There are two basic settings for the tests. In other words they are two requirements on the syntax of tests, so that tests are executed properly. As mentioned previously, one of them is that we have a preset label prefix for recognizing all tests in the Alloy model. The other is that, in order for Alloy parser to recognize function calls, all tests should have curly braces around the predicates after “run”. Otherwise, the predicate will be inlined, i.e., replaced with its body, and we will not be able to know that the command calls the predicate.

### RUNNING THE TESTS

To test the model, we can choose to run all tests in the Execution tab and all functionality for AUnit testing will be executed. This part shows where “Execute All Tests” choice comes from and how clicking it will do the job.

Partial code of drop down menu in Execution tab is shown in Fig 4.1. The number of test cases are counted before the menu is loaded. As long as at least one test exists, there will be an item called “Execute All Tests” in Execution menu, which allows user to run all tests and see the results. When the listener of “Execute all Tests” gets clicking action, index -6 is fed to method “doRun” to run all tests. Method “doRun” inside the class SimpleGUI is responsible for running all commands inside the module. It takes an index to indicate which task to fulfill. A non-negative index indicates running the corresponding command of that index in order. For example “0” means running the first command. And index “-1” means running all commands. We added index “-6” as the one that indicates running all tests. Given the index, “doRun” method creates a task and hands

it over to WorkerEngine. WorkerEngine then will run the task by invoking the “run” method of the task, as shown in Fig 4.2.

```
if (tests.size() >= 1) {
    JMenuItem y = new JMenuItem("Execute All Tests", null);
    y.setMnemonic(VK_A);
    y.addActionListener(doRun(-6));
    popup.add(y, 1);
}
```

Fig 4.1 Invoking doRun(-6)

```
440 /** Task that perform one command. */
441 static final class SimpleTask1 implements WorkerTask {
442     private static final Long serialVersionUID = 0;
443     public A4Options options;
444     public String tempdir;
445     public boolean bundleWarningNonFatal;
446     public int bundleIndex;
447     public int resolutionMode;
448     public Map<String,String> map;
449     public SimpleTask1() { }
450     public void cb(WorkerCallback out, Object... objs) throws IOException { out.callback(objs); }
451     public void run(WorkerCallback out) throws Exception {
452         cb(out, "S2", "Starting the solver...\n\n");
453         StringBuilder coverage = new StringBuilder();
454         StringBuilder postcover = new StringBuilder();
455         StringBuilder precover = new StringBuilder();
456         final SimpleReporter rep = new SimpleReporter(out, options.recordKodkod);
457         final Module world = Computil.parseEverything_fromFile(rep, map, options.originalFilename, resolutionMode);
458         final List<Sig> sigs = world.getAllReachableSigs();
459         final ConstList<Command> cmds = world.getAllCommands();
460         cb(out, "warnings", bundleWarningNonFatal);
461         if (rep.warn>0 && !bundleWarningNonFatal) return;
462         List<String> result = new ArrayList<String>(cmds.size());
463         if (bundleIndex==2) {
464             result.add(" ");
465         } else if (bundleIndex == -6) {
466             cb(out, "bold", "Executing all tests...\n\n");
467         }
468     }
469 }
```

Fig 4.2 Method “run” in SimpleTask1

## EVALUATION AND COVERAGE

As mentioned previously, in an instance of the model, each signature, field or expression have a set of atoms, and each predicate or formula is either true or false. Atoms are actual values for expression, signature and fields. This section explains the way we determine what the instance has for the entities and the way to keep coverage status.

We use evaluators for getting real values for all entities inside the module. Evaluator is one of the APIs Alloy Analyzer provide for users and developers to do further exploration with instances. There is one evaluator associated with each instance

found. Given an expression, a signature or a field, it can give us a set of atoms that the instance holds. Given a formula it can tell whether the formula evaluates to true or false. To test the model, we just find all entities to evaluate, get all instances generated by tests, grab the evaluator for each instance and evaluate all entities use all evaluators and update coverage status along the process.

To keep track of the coverage status, we have an integer associated with every signature and field, as well as all predicates, facts, assertions and all primitive Booleans formulas and quantified formulas. The lowest  $n$  bits of the integer represent the  $n$  requirements for each entity. The integer representing coverage status is initially 0. Every time a requirement is met for an instance, we do bit operation “or 1” on the bit associated with that requirement. So that it is updated to 1 if it was 0 or remains to be 1 if it was 1.

As for a test requirement, as long as one of the instance meets the requirement, that requirement is regarded as covered. For example, for a signature  $S$ , the instance of a test has one actual value ( $S\$0$ ) for  $S$ , then the requirement  $|S| = 1$  is covered.

In this implementation, we specifically dealt with signatures, fields and predicates. The rest will be part of future works.

## **GENERAL IMPLEMENTATION**

Having provided an overview of an Alloy model and a complete test criterion for Alloy, we now turn our attention to the details of test coverage. Note that the structure of Alloy models can be regarded as trees, which are usually complicated. We need to extract all signatures, fields, facts, assertions and predicates and evaluate them with every instance we find by running our tests. For signatures and fields, the task of evaluating with every instance is relatively simple. We just need to know the number of actual

values they have for each instance, so that we know which requirement is covered and which is not. To do that, we use the evaluators mentioned above.

Predicates, facts and assertions are a bit more complex. Same as signatures and fields, they are associated with a coverage status and evaluated by evaluators. The complicated part is that they have a body that is a tree of expressions and formulas. Especially when a quantified formula is involved, the data structure of the tree is needed for the evaluation and coverage status updating.

Consider predicates for example. We have a list of predicates and each predicate has a list of primitive Booleans and quantified formulas. We evaluate a predicate with all tests that called it at least once. At the same time we also evaluate all primitive Booleans and quantified formulas in it. Primitive Booleans are easy to deal with. It has only two coverages: true and false. As for quantified formulas, we have a specific data structure designed for it: a tree for quantified formula evaluation.

### **TREE FOR QUANTIFIED FORMULA EVALUATION**

The difficulty that lies in evaluating quantified formula is that it is a tree like data structure. Its children's evaluation relies on the variable declaration of the parent. Thus we need to keep the hierarchy information. Extracting all formulas to be evaluated inside it, like what we do with signatures and fields of the whole module, does not work. The code for this part will be included in Appendix.

QtTestTree is a tree like datastructure we introduced to evaluate quantified formulas and all primitive Booleans inside it. It has an attribute, a couple methods and several static nested classes in it to help it achieve its job. They will be explained one by one in the following paragraphs.

The basic ideas of this data structure are as follow:



- 1) We maintain the hierarchy relationship of the formulas inside this tree.
- 2) To replace variables with their atoms, we do it recursively through the tree until we hit leaves.
- 3) At the leaves of the tree we do string replacement to replace the variables with their atoms, since there will be no more variable declaration inside the leaves.
- 4) To evaluate a node of the tree, we construct the latest pars-able string version of the formula of the node by calling toString method recursively, parse it into a valid Expr object and feed it into the evaluator.
- 5) Each leaf keeps a stack of previous version of itself. It is for the purpose of maintaining the information about previous replacement and being able to revert back so that we can replace the same variable with another atom. For example, consider “Q x1:d | Q x2:d | some (x1 & x2)”. Domain d has two atoms: X\$0 and X\$1. The declared variables “x1” and “x2” are of the same domain and “some (x1 & x2)” is a primitive Boolean formula. After variable replacement in both quantified formulas, the stack in the wrapper of the primitive Boolean formula is [“some X\$0 & x2”, “some X\$0 & X\$0”]. After evaluating the formula being “some X\$0 & X\$0”, we need to replace x2 with X\$1 and evaluate it again. If we do not keep a stack, we will not be able to tell which X\$0 is originally x2. Since we have the stack, all we need to do is to pop out the last item in the stack after evaluation and do the variable replacement on the last item of the stack next time.

To further explain the details, first we start with the nested classes. There are five of them, named “QtExp”, “QtPrimBool”, “QtExpList”, “QtNode” and “QtVar”. All nested classes are wrappers for some original data structure of Alloy Analyzer. They designed to be compact and only for testing. QtExp is a wrapper for Expr, a base class in Alloy Analyzer. QtExp is the base class for QtPrimBool, QtExpList and QtNode.

QtPrimBool is the wrapper for all data structure that are primitive Booleans. QtExpList is the wrapper for ExprList and QtNode is the wrapper for ExprQt, which is the class for quantified formulas in Alloy Analyzer. It is obvious that only QtPrimBool and QtNode need coverage status because they are the only two classes that has coverage requirements. Also instances of QtPrimBool is always the leave of the tree. So is the domain of variable declaration inside the quantified formulas. To make a clear explanation, we look at QtVar first.

QtVar is the simplest among all these nest classes. It serves as the wrapper of variable declaration inside the quantified formula. It has only two attributes. The label of the variable that is a string and the domain of the variable that is a QtExp. To make this happen, QtExp is a concrete class though it mostly serves as the base class of other classes. Since there will be no quantified formula or primitive Boolean formula inside of the domain, we do not need to look inside it. Thus QtExp is the only other type that is regarded as leaves other than QtPrimBool.

QtNode, as explained, is the wrapper for a quantified formula and is a subclass of QtExp. It has four attributes in addition to what QtExp has: operator “op”, variable declaration “var”, “body” and “coverageStatus”. The operator “op” is used only in toString() to make the latest pars-able string version of the quantified formula. Variable declaration is of type QtVar as mentioned and body is of type QtExpList. To evaluate the QtNode, we do follow this algorithm:

- 1) Get the list of atoms for the variable declared, say v, by evaluating its domain.
- 2) If the list is empty, the first testing requirement is covered.
- 3) If the list is not empty, iterate through the atoms. In every iteration,

- a. Replace all  $v$  in the body of the formula with the current atom we are considering. It will push a new item into the stack of the leaves.
- b. Evaluate the body of the formula and keep the result.
- c. Call the evaluation method of the body so that the sub nodes of the body will be evaluated recursively with current replacement in place.
- d. Pop out the last replacement inside the stack of the leaves.

4) Update the coverage status according to all evaluation results of the body.

These procedures are handled by the eval method. The whole process is a backtracking algorithm.

The syntax of quantified formula allows duplicate names, which means variables declared inside the quantified formula or its body can have same names as each other. For example “ $Q x:d1 \text{ some } x \text{ and } x:d2 \text{ no } x$ ” is a legitimate quantified formula. Variables are bind to the latest declaration of the same label. In our example,  $x$  in “some  $x$ ” is of domain  $d1$  and  $x$  in “no  $x$ ” is of domain  $d2$ . The variable replacement method of QtNode works as follows:

- 1) Check whether the variable to be replaced has the same label as its own declared variable.
- 2) If not, it will call replaceSelf on its body and variable domain.
- 3) If so, no replacement will happen in its body and domain of its declared variable. But it will call method dupeone. The dupeone method will recursively call itself on the formula’s subnodes and insert a string into the stack of the leaves. The string is the same as the last item in the stack. So it can be popped after evaluation.

QtExpList has an operator “op” and a list of QtExp “exprs”. The operator “op”, similarly to the one in QtNode, is only for the use of generating pars-able strings that represents the formula. All its methods recursively call the same method in the items of its QtExp list and use the result if they return anything.

QtExp keeps the original expression as well as a stack of string that we use for variable replacement. It has getLast, popLast and dupeone methods for maintaining the stack. It also has replaceSelf method to generate the latest version of string representation. There also is a toString method that returns the latest string representation. It has other three empty methods: eval, getUncoveredReq4SelfAndSub and getUncoveredReq4SelfAndSubNum. They are used in its subclasses and their names are self-explanatory. Method eval evaluates the current node, recursively call the eval method of its subnodes and updates coverage status if necessary. Method getUncoveredReq4SelfAndSub gets uncovered testing requirements for itself and its subnodes recursively. Method getUncoveredReq4SelfAndSubNum collects the number of uncovered requirements of itself and its subnodes recursively. All methods of QtExp are override by their subclasses if necessary.

QtPrimBool keeps a coverage status and inherits the functionality of variable replacement from QtExp. It also overrides eval, getUncoveredReq4SelfAndSub and getUncoveredReq4SelfAndSubNum methods.

In addition to all these nested classes, QtTestTree has an attribute called “head” that is of type QtNode and three methods: eval, getUncoveredReq and getUncoveredReqNum. These methods call the relavent methods in “head” to do the jobs recursively. Method eval() is used to run evaluation and update coverage status for all primitive Booleans and quantified formulas in the tree. It calls the method eval() on its head and recursively executes the evaluation task. Method getUncoveredReq() returns a

string with all uncovered test requirements in it. It also calls a method on the head that computes the uncovered requirements recursively. Method `getUncoveredReqNum()` gives the number of all uncovered requirements.

To conclude, every time a first level quantified formula is encountered, an instance of `QtTestTree` is constructed with it. The phrase “first level” means that the quantified formula must not be nested in another one. As discussed above, after the tree is instantiated, once the method `eval` is called, it will recursively call the `eval` method on its “head”. Recursively, all `QtNodes` will evaluate their variable domains to get all possible values for variables declared. Possible values for variables are also known as atoms. The `QtNode` will replace the variables in its body that are bound to its declared variables with one of the atoms and then compute truth-value of its body. After that, it will call `eval` method on its body. After the call returned, it will revoke the replacement of variables to previous status. This procedure is repeated for all atoms. This procedure finds out which requirement is covered for this formula. The coverage status of this node is then updated. As a part of the body of a quantified formula, all primitive Booleans compute their truth-values when the `eval` method is called and update coverage status. Every time a variable replacement is executed, it finds the leaves and performs string replacement. To evaluate a formula or an expression, it gets the latest version of string representation of that expression or formula by calling `toString` method. Then parses the string to a legitimate expression and evaluates it using evaluator.

## **Chapter 5: Discussion and Future Work**

### **COVERAGE FOR TESTS WITH NO INSTANCE**

As mentioned previously, in our implementation only tests that have instances contributes to the coverage. However, those without an instance still help finding the defects of the model. If we don't have any test that cannot have an instance, all facts in the model are always true in the tests. That does not complete the coverage. It is more reasonable and also consistent with JUnit setting to let them contribute to the coverage too.

The way to solve this is to relax the constrain introduced by facts and assertions when calculating coverage. After running the tests and before calculating coverage, we change all facts and assertions into predicates, so that they don't constrain the model. Then we run the tests again, we should have an instance for all tests and will be able to calculate the coverage more comprehensively.

### **A BETTER DATA STRUCTURE**

According to the grammar of Alloy [4], an entire Alloy module is a tree like data structure. However, our implementation flattens most of the structure except for quantified formulas. To implement testing for other entities, like let expression, we have to build other data structures that works only for them, which is not generic at all. As a result, the coverage for facts and assertion are still remained blank as well as for relationship declaration and let expression.

A better way to implement this is to construct wrappers for original data structure of Alloy Analyzer. Regard the whole module as a tree and build a comprehensive data structure that embrace everything. It would be more comprehensive and reasonable. It will also give a more robust performance.

The data structure should be able to replace some of its nodes with manually constructed expressions. So that we can replace facts with equivalent predicates, reconstruct quantified formulas with multiple declaration into nested style and replace variables with atoms for evaluation. The data structure should be able to evaluate itself recursively and recursively report uncovered requirements. Also after replacing nodes, the operation of replacement should be able to be reverted back easily. That would allow the backtracking algorithm.

Additional future work can integrate our tool with other test automation tools for Alloy, e.g., MuAlloy [6], which provides mutation testing for Alloy.

## **Chapter 6: Conclusion**

This report gives a prototype for implementation of AUnit, a test automation framework that was introduced in previous work on automated testing for Alloy. It has all core functionalities for unit tests that can distinguish tests from other commands, run all tests, show the result including the number of tests run, the number of tests failed, coverage status and coloring, all test requirements and all uncovered requirements. It can compute coverage for signatures, fields, predicates and specifically for primitive Booleans and quantified formulas. It can allow users to check the quality of their models in the spirit of traditional unit testing.



## Appendix

### CODE OF QTTESTTREE

```
package edu.mit.csail.sdg.alloy4whole;

import edu.mit.csail.sdg.alloy4.WorkerEngine.WorkerCallback;
import edu.mit.csail.sdg.alloy4compiler.ast.Browsable;
import edu.mit.csail.sdg.alloy4compiler.ast.Decl;
import edu.mit.csail.sdg.alloy4compiler.ast.Expr;
import edu.mit.csail.sdg.alloy4compiler.ast.ExprBinary;
import edu.mit.csail.sdg.alloy4compiler.ast.ExprCall;
import edu.mit.csail.sdg.alloy4compiler.ast.ExprHasName;
import edu.mit.csail.sdg.alloy4compiler.ast.ExprITE;
import edu.mit.csail.sdg.alloy4compiler.ast.ExprLet;
import edu.mit.csail.sdg.alloy4compiler.ast.ExprList;
import edu.mit.csail.sdg.alloy4compiler.ast.ExprQt;
import edu.mit.csail.sdg.alloy4compiler.ast.ExprUnary;
import edu.mit.csail.sdg.alloy4compiler.ast.ExprVar;
import edu.mit.csail.sdg.alloy4compiler.ast.Sig;
import edu.mit.csail.sdg.alloy4compiler.ast.Sig.Field;
import edu.mit.csail.sdg.alloy4compiler.ast.Module;
import edu.mit.csail.sdg.alloy4compiler.parser.CompUtil;
import edu.mit.csail.sdg.alloy4compiler.translator.A4Solution;
import edu.mit.csail.sdg.alloy4compiler.translator.A4Tuple;
import edu.mit.csail.sdg.alloy4compiler.translator.A4TupleSet;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.SimpleFormatter;
import java.util.Map;
import java.io.IOException;

public final class QtTestTree {

    QtNode head;

    public QtTestTree (ExprQt qt) throws Exception {
```

```

    head = new QtNode(qt);
}

public QtTestTree (QtNode head_) {
    head = head_;
}

public void eval(Module world, A4Solution ai) throws Exception {
    for(ExprVar a:ai.getAllAtoms()) { world.addGlobal(a.label,
a); }
    for(ExprVar a:ai.getAllSkolems()) { world.addGlobal(a.label, a); }
    head.eval(world, ai);
}

public String getUncoveredReq() {
    return head.getUncoveredReq4SelfAndSub();
}

public int getUncoveredReqNum() {
    return head.getUncoveredReq4SelfAndSubNum();
}

public static String replace(String original, String var, String
value) {
    int ind = original.indexOf(var);
    while (ind != -1) {
        if (nextIsComma(original, ind + var.length())) break;
        if (!notReplace(original, ind, ind + var.length())) original =
original.substring(0, ind) + value + original.substring(ind +
var.length());
        ind = original.indexOf(var, ind + var.length());
    }
    return original;
}

private static boolean notReplace(String original, int sta, int end)
{
    if (sta - 1 >= 0 && (Character.isLetter(original.charAt(sta - 1))
|| original.charAt(sta - 1) == '_')) return true;
    if (end < original.length() &&
(Character.isLetter(original.charAt(end)) || original.charAt(end) ==
'_')) return true;
    return false;
}
}

```

```

private static boolean nextIsComma(String original, int ind) {
    if (ind >= original.length()) return false;
    while(original.charAt(ind) == ' ') ind++;
    if (original.charAt(ind) == ':') return true;
    return false;
}

public static boolean isPrimBool(Expr curExpr) {
    if ((curExpr instanceof ExprBinary || curExpr instanceof ExprCall
||
    curExpr instanceof ExprITE || curExpr instanceof
ExprLet ||
    curExpr instanceof ExprUnary || curExpr instanceof
ExprVar)
        && curExpr.type().is_bool) return true;
    return false;
}

//=====

public class QtExp {
    Expr exp;
    List<String> stack;

    public QtExp(Expr expr) {
        exp = expr;
        stack = new ArrayList<>();
        stack.add(expr.toString());
    }

    public String getLast() {
        return stack.get(stack.size() - 1);
    }

    public void dupeone() {
        String a = new String(getLast());
        stack.add(a);
    }

    public void popLast() {
        stack.remove(stack.size() - 1);
    }

    public void replaceSelf(QtVar var, String atom) {

```

```

        String replaced = QtTestTree.replace(getLast(), var.label,
atom);
        stack.add(replaced);
    }

    public void eval(Module world, A4Solution ai) throws Exception {};
    public String getUncoveredReq4SelfAndSub() {
        return "";
    }
    public int getUncoveredReq4SelfAndSubNum() { return 0;}

    public String toString() {
        return stack.get(stack.size() - 1);
    }
}

//-----

public class QtPrimBool extends QtExp {
    int coverageStatus;
    public QtPrimBool(Expr expr) throws Exception {
        super(expr);
    }

    public String getUncoveredRequirements() {
        StringBuilder sb = new StringBuilder();
        if ((coverageStatus & 1) == 0) { sb.append(exp.toString());
sb.append(" = true\n");}
        if ((coverageStatus & 2) == 0) { sb.append(exp.toString());
sb.append(" = false\n");}
        return sb.toString();
    }

    public String getAllRequirements() {
        StringBuilder sb = new StringBuilder();
        sb.append(exp.toString()); sb.append(" = true\n");
        sb.append(exp.toString()); sb.append(" = false\n");
        return sb.toString();
    }

    public int getUncoveredRequirementsNum() {
        int count = 0;
        if ((coverageStatus & 1) == 0) count++;
        if ((coverageStatus & 2) == 0) count++;
        return count;
    }
}

```

```

    }

    public int getAllRequirementsNum(){
        return 2;
    }

    public String getUncoveredReq4SelfAndSub() {
        return getUncoveredRequirements();
    }
    public int getUncoveredReq4SelfAndSubNum() { return
getUncoveredRequirementsNum();}

    public void eval(Module world, A4Solution ai) throws Exception {
        Expr exp = CompUtil.parseOneExpression_fromString(world,
toString());
        boolean bool = (Boolean)ai.eval(exp);
        if (bool) coverageStatus |= 1;
        else coverageStatus |= 2;
    }
}

//-----

public class QtExpList extends QtExp {
    ExprList.Op op;
    List<QtExp> exprs;

    public QtExpList(Expr exp) throws Exception {
        super(exp);
        exprs = new ArrayList<>();
        if (exp instanceof ExprList) make((ExprList)exp);
        else {
            if(QtTestTree.isPrimBool(exp))exprs.add(new QtPrimBool(exp));
            else exprs.add(new QtNode((ExprQt)exp));
        };
    }

    public void popLast() {
        for(int i=0; i<exprs.size(); i++) exprs.get(i).popLast();
    }

    public void dupeone() {
        for(int i=0; i<exprs.size(); i++) exprs.get(i).dupeone();
    }
}

```

```

public String getUncoveredReq4SelfAndSub(){
    StringBuilder sb = new StringBuilder();
    for (QtExp expr : exprs) {
        sb.append(expr.getUncoveredReq4SelfAndSub());
    }
    return sb.toString();
}

public int getUncoveredReq4SelfAndSubNum() {
    int count = 0;
    for (QtExp expr :exprs) {
        count += expr.getUncoveredReq4SelfAndSubNum();
    }
    logger.log(Level.INFO, count + "");
    return count;
}

public void eval(Module world, A4Solution ai) throws Exception {
    for(int i=0; i<exprs.size(); i++) exprs.get(i).eval(world, ai);
}

public void replaceSelf(QtVar var, String atom) {
    for(int i=0; i<exprs.size(); i++) exprs.get(i).replaceSelf(var,
atom);
}

public String toString() {
    StringBuilder out = new StringBuilder();
    if (op == null) out.append(exprs.get(0).toString());
    else if (op == ExprList.Op.DISJOINT || op ==
ExprList.Op.TOTALORDER){
        out.append(op).append("[");
        for(int i=0; i<exprs.size(); i++) { if (i>0)
out.append(", "); out.append(exprs.get(i).toString()); }
        out.append(']');
    } else {
        out.append("(");
        for(int i=0; i<exprs.size(); i++) { if (i>0)
out.append(" " + op.toString().toLowerCase() + " ");
out.append(exprs.get(i).toString()); }
        out.append(")");
    }
    return out.toString();
}
}

```

```

}

//-----

public class QtNode extends QtExp {
    ExprQt.Op op;
    QtVar var;
    QtExpList body;
    int coverageStatus;
    public QtNode(ExprQt qt) throws Exception {
        super(qt);
        op = qt.op;
        qt = restructQt(qt);
        var = new QtVar(qt.decls.get(0).names.get(0).toString(),
            ((ExprUnary)(qt.decls.get(0).expr)).sub);
        body = new QtExpList(qt.sub);
    }

    private ExprQt restructQt(ExprQt curExpr) {
        if (curExpr.decls.size() > 1) {
            // reformat
            int tail = curExpr.decls.size() - 1;
            Expr newSub = curExpr.sub;
            while (tail > 0) {
                newSub = curExpr.op.make(curExpr.pos,
                    curExpr.closingBracket, curExpr.decls.subList(tail, tail + 1), newSub);
                tail--;
            }
            curExpr = (ExprQt)curExpr.op.make(curExpr.pos,
                curExpr.closingBracket, curExpr.decls.subList(tail, tail + 1), newSub);
        }
        Decl d = curExpr.decls.get(0);
        if (d.names.size() > 1) {
            List<Decl> ds = new ArrayList<>();
            for (int i = 0; i < d.names.size(); i++) {
                ds.add(new Decl(d.isPrivate, d.disjoint, d.disjoint2,
                    d.names.subList(i, i + 1), d.expr));
            }
            int tail = ds.size() - 1;
            Expr newSub = curExpr.sub;
            while (tail > 0) {
                newSub = curExpr.op.make(curExpr.pos,
                    curExpr.closingBracket, ds.subList(tail, tail + 1), newSub);
                tail--;
            }
        }
    }
}

```

```

        curExpr = (ExprQt)curExpr.op.make(curExpr.pos,
curExpr.closingBracket, ds.subList(tail, tail + 1), newSub);
    }
    logger.log(Level.INFO, curExpr.toString());
    return curExpr;
}

public void popLast() {
    var.domain.popLast();
    body.popLast();
}

public void dupeone() {
    var.domain.dupeone();
    body.dupeone();
}

public String getUncoveredRequirements() {
    return getUncoveredRequirementsFromStatus(coverageStatus);
}

public String getAllRequirements() {
    return getUncoveredRequirementsFromStatus(0);
}

private String getUncoveredRequirementsFromStatus(int status) {
    String domain = this.var.domain.exp.toString();
    String b = this.body.exp.toString();
    StringBuilder sb = new StringBuilder();
    if ((status & 1) == 0) { sb.append("#"); sb.append(domain);
sb.append(" = 0\n");}
    if ((status & 2) == 0) { sb.append("#"); sb.append(domain);
sb.append(" = 1 and "); sb.append(b); sb.append(" = true\n");}
    if ((status & 4) == 0) { sb.append("#"); sb.append(domain);
sb.append(" = 1 and "); sb.append(b); sb.append(" = false\n");}
    if ((status & 8) == 0) { sb.append("#"); sb.append(domain);
sb.append(" >= 2 and "); sb.append(b); sb.append(" = true for all var
in the domain\n"); }
    if ((status & 16) == 0) { sb.append("#");
sb.append(domain); sb.append(" >= 2 and "); sb.append(b); sb.append(" =
false for all var in the domain\n"); }
    if ((status & 32) == 0) { sb.append("#");
sb.append(domain); sb.append(" >= 2 and "); sb.append(b); sb.append("
is false for at least one var in the domain \n    and also is true for
at least one var in the domain\n"); }

```



```

    return sb.toString();
}

public int getUncoveredRequirementsNum() {
    int count = 0;
    if ((coverageStatus & 1) == 0) count++;
        if ((coverageStatus & 2) == 0) count++;
        if ((coverageStatus & 4) == 0) count++;
        if ((coverageStatus & 8) == 0) count++;
        if ((coverageStatus & 16) == 0) count++;
        if ((coverageStatus & 32) == 0) count++;
    return count;
}

public int getAllRequirementsNum(){
    return 6;
}

public String getUncoveredReq4SelfAndSub(){
    StringBuilder sb = new StringBuilder();
    sb.append(getUncoveredRequirements());
    sb.append("-----In this Qt-----\n");
    sb.append(body.getUncoveredReq4SelfAndSub());
    return sb.toString();
}

public int getUncoveredReq4SelfAndSubNum() {
    int count = getUncoveredRequirementsNum();
    count += body.getUncoveredReq4SelfAndSubNum();
    logger.log(Level.INFO, count + "");
    return count;
}

public void replaceSelf(QtVar variable, String atom) {
    var.domain.replaceSelf(variable, atom);
    if (!var.label.equals(variable.label))
body.replaceSelf(variable, atom);
    else body.dupeone();
    // replaced = toString();
}

public void eval(Module world, A4Solution ai) throws Exception {
    Expr exp = CompUtil.parseOneExpression_fromString(world,
var.domain.toString());
    List<String> values = new ArrayList<>();

```

```

    for (A4Tuple s : (A4TupleSet)ai.eval(exp)) {
        values.add(s.toString());
    }
    if (values.size() == 0) coverageStatus |= 1;
    else if (values.size() == 1){
        body.replaceSelf(var, values.get(0));
        Expr bodyExp =
CompUtil.parseOneExpression_fromString(world, body.toString());
        boolean bodyVal = (Boolean)ai.eval(bodyExp);
        if (bodyVal) coverageStatus |= 2;
        else coverageStatus |= 4;
        body.eval(world, ai);
        body.popLast();
    } else {
        boolean hasTrue = false;
        boolean noFalse = true;
        for (int i = 0; i < values.size(); i++) {
            body.replaceSelf(var, values.get(i));
            Expr bodyExp =
CompUtil.parseOneExpression_fromString(world, body.toString());
            boolean bodyVal = (Boolean)ai.eval(bodyExp);
            hasTrue = hasTrue || bodyVal;
            noFalse = noFalse && bodyVal;
            body.eval(world, ai);
            body.popLast();
        }
        if (hasTrue && noFalse) coverageStatus |= 8;
        else if (!hasTrue && !noFalse) coverageStatus |= 16;
        else coverageStatus |= 32;
    }
}

public String toString() {
    return "(" + op + " " + var.toString() + " | " + body.toString()
+ ")";
}
}

//-----

public class QtVar {
    String label;
    QtExp domain;
}

```

```
public QtVar(String label, Expr domain) throws Exception {
    this.label = label;
    this.domain = new QtExp(domain);
}

public String toString() {
    return label + " : " + domain.toString();
}
}
```

## References

- [1] Allison Sullivan, Razieh Nokhbeh Zaeem, Sarfraz Khurshid, and Darko Marinov (2014). Towards a test automation framework for Alloy. In Proc. *International SPIN Symposium on Model Checking of Software*.
- [2] Alloy - frequently asked questions website. Retrieved May 03, 2017, from <http://alloy.mit.edu/alloy/faq.html>
- [3] Ammann, P., & Offutt, J. (2016). *Introduction to Software Testing*. Cambridge University Press.
- [4] Jackson, D. (2012). *Software Abstractions: logic, language, and analysis*. MIT press.
- [5] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem and Sarfraz Khurshid (2017). Automated Test Generation and Mutation Testing for Alloy. In Proc. *10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*.
- [6] Wang, K. (2015). *MuAlloy: An automated mutation system for Alloy*. Masters thesis. University of Texas at Austin.