2018 IEEE 11th International Conference on Software Testing, Verification and Validation

AUnit: A Test Automation Tool for Alloy

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

***Abstract*—Software models help improve the reliability of software systems: models can convey requirements, and can analyze design and implementation properties. A key strength of Alloy, a commonly used first-order modeling language, is the Alloy Analyzer tool-set. The Analyzer allows users to execute commands over models by leveraging a fully automatic SAT-based analysis engine. However, prior to the introduction of AUnit – a testing framework for Alloy – users had to rely on ad-hoc practices to validate their models. In this paper, we present our efforts to establish a formal testing environment in the Alloy Analyzer by creating an AUnit extension. We present additional grammar to support test case creation, as well as the details for executing test suites, calculating test suite coverage, and automatically generating test suites. The tool is available as a stand-alone executable at the following URL (https://sites.google.com/view/aunitanalyzer).**

I. Introduction

In today's society, we are becoming increasingly dependent on software systems. However, we also constantly witness the negative impacts of buggy software. One way to help develop better software systems is to leverage software models. When forming requirements, software models can be used to clearly communicate to all stakeholders both the desired system as well as the environment it will be deployed in. When creating designs and implementations, software models can help reason over how well the design and implementation choices satisfy the requirements. As such, software models can help detect flaws earlier in development and thus aid in the delivery of more reliable systems.

Alloy, a first-order declarative language based on relations, is a well-known modeling language [1]. Alloy is designed to be (1) easy to understand; and (2) amenable to automatic analysis. One of the main benefits of Alloy is its integrated development environment (IDE) based on the Alloy Analyzer. The Analyzer provides an automated analysis environment where developers can execute commands over a model, which the Analyzer resolves by performing scope-bounded analysis using off-the-shelf SAT solvers. As output, the Alloy Analyzer reports satisfying assignments to the sets and relations of the model such that the command's formulas are true, known as *instances*. Users can inspect these instances textually or graphically, and evaluate expressions and formulas over these instances using the Evaluator functionality [12].

As a lightweight formal method backed by an instance enumerating tool-set, Alloy helps developers understand their system by displaying the consequences, intended or not, of their specifications, helping identify any missing or incorrect properties, and exploring the impact of modifications. As such, Alloy has been utilized in multiple domains, e.g. debugging UML diagrams [5], [4], analyzing security protocols in fields such as firewalls and HTML [6], [7], and modeling network switches [3]. Notable recent work by Pamela Zave uses Alloy to reveal flaws in the Chord ring-maintenance protocol [15], which has influenced systems in industry.

However, the expected benefits of formal modeling and analysis are reduced quite a bit if the model itself is incorrect. In our experience, this is a valid risk. While Alloy offers succinct formulation of complex properties, Alloy's expressive operators (transitive closure, quantified formulas) can make writing non-trivial properties challenging, especially for beginner users. Traditionally, developers gain confidence in their code through testing. However, traditional testing frameworks are designed for imperative, but not declartive, languages. Consider the declarative execution environment for Alloy. When a command is executed, there is no traditional notion of where the "execution" starts, how it proceeds, what conditional branches it encounters, what path it takes, and how the values of the program variables are updated or the final return values computed. Instead, the SAT back-end looks for satisfying instances to the propositional formula based on the command executed by the user.

Our recent work introduced AUnit [11], [8] for testing Alloy models in the spirit of unit testing imperative code. AUnit defines what a test case is, what test execution and outcome are, and what the coverage criteria are in the context of declarative models in Alloy. We wanted our test case structure to allow a developer to directly test for expected behavior, i.e. that a specific instance is properly generated (or prevented) by any given formula. Therefore, AUnit test cases consist of two key components: (1) a valuation used to define a specific instance to be evaluated and (2) a command used to identify the model formulas under test. A test *passes* if the valuation is a valid instance of the command; otherwise, the test fails. AUnit has served as a foundation for integrating well established traditional testing techniques into Alloy such as automated test generation [10] and mutation testing [10], [13]. AUnit has also led to program synthesis techniques [9].

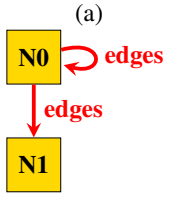
Prior work has looked to provide a basic AUnit prototype built around the existing Analyzer's API and noted the limitations encountered [14]. This tool paper focuses on our efforts to implement AUnit as an extension to the Alloy Analyzer, modifying the grammar and building new functionality and user interfaces into the Analyzer as needed. The remainder of the paper is outlined as follows. First, we introduce an Alloy model of a directed tree and an associated test suite to present

978-1-5386-5012-7/18/$31.00 ©2018 IEEE
DOI 10.1109/ICST.2018.00047

Fig. 1: AUnit Test Case

($Test0$)

(a)



(b)

```
1  pred Val0 {
2    some disj N0, N1 : Node {
3      Node = N0 + N1
4      edges = N0->N0 + N0->N1
5    }
6  }
```

(c)

```
Test0: run { Val0[] and !isDirectedTree[] }
```

essential background concepts. Then, we highlight changes to the Analyzer's grammar to support the core AUnit functionality. Next, we demonstrate how to use these features to create, execute and calculate coverage for a test suite. We also outline support for $AGen_{Cov}$, a coverage-directed input generation technique. Support for $AGen_{Cov}$ is included to automate input generation in order to help facilitate a comprehensive testing environment for Alloy. Lastly, we highlight the lessons we learned adapting our research into a prototype tool.

II. Background

In this section, we will first introduce an illustrative, but simple, example Alloy model. Then, we will step over: (1) an example test case, (2) the corresponding coverage for that test case and (3) a test case generated using $AGen_{Cov}$. Further details about AUnit can be found in [11] while further details about $AGen_{Cov}$ can be found in [10].

A. Example Model: Directed Tree

Consider the following model of a directed tree:

```
1  sig Node { edges: set Node }
2  pred isDirectedTree {
3      no iden & ^edges --acyclic
4      edges.~edges in iden --injective
5      Node->Node in ^(edges + ~edges) --connected
6  }
```

Line 1 encompasses a signature ('sig') paragraph, that introduces the named set Node. Node contains one binary relation "edges", which relates each Node to one or more other Node elements, forming the edge(s) of the tree. From an object oriented perspective, this can be thought of as declaring a Node class with an edges attribute. Lines 2 - 6 outlines the predicate ('pred') paragraph "isDirectedTree" that contains the logic for the model to produce only directed trees: line 3 outlines acyclicity to prevent self-loops, line 4 forces edges to be a one to one function, and line 5 ensures any element in Node is connected to the tree. These statements are achieved using a subset of the Alloy syntax, specifically: set multiplicity ('no'), the identity relation ('iden'), set intersection ('&'), transitive closure ('^'), relational join ('.'), transpose (' '), subset ('in'), relational product ('->') and set union ('+'). The complete Alloy grammar can be found at [2].

Now, consider what test cases we might create for this model. For instance, we could start by checking if a cyclic tree is prevented by using the test case outlined in Figure 1. The valuation is given both graphically (a) and textually (b), while the command is listed on the bottom (c). Note that a valuation simply outlines the shape of an instance. If the valuation were to be executed, there is only 1 possible instance that can be generated. Running "Test0" returns a passing result, increasing our confidence that isDirectedTree generates acyclic trees. Notice, our command is "!isDiredtedTree". Using the negation of a predicate indicates that we expect the valuation to *not* be generated by the predicate. The command can be any valid Alloy command.

TABLE I: Coverage provided by Test0

Construct	Coverage
Node	$\|s\| \geq 2$
edges	$\|r\| \geq 2$
no iden & ^edges	f = false
iden & ^edges	$\|e\| = 1$
iden	$\|e\| \geq 2$
^edges	$\|r\| \geq 2$
edges.~edges in iden	f = true
edges.~edges	$\|e\| = 1$
Node->Node in ^(edges + ~edges)	f = true
Node->Node	$\|e\| \geq 2$
^(edges + ~edges)	$\|e\| \geq 2$
edges + ~edges	$\|e\| \geq 2$
~edges	$\|e\| \geq 2$

AUnit also supports declaring *partial* valuations in which the user outlines some sets and relations in the model, and the SAT solver defines the rest. Partial valuations can lead to multiple instances when executed. For example, we could write the following test case to check our connectivity formula:

```
1  pred Val1 { !(lone Node) and no edges }
2  Test1: run {Val1[] and !isDirectedTree[]} for 5
```

In this case, our valuation outlines any instance in which there are at least 2 Node elements and an empty edges relation. Our command then checks that all such instances are invalid directed trees. Note, our command now specifies a *scope* ('for 5'). The Alloy Analyzer uses the scope to provide an upper bound on the universe of discourse to the SAT solver. All commands are run with a default scope of 3, unless the user specifies otherwise.

AUnit also introduces coverage criteria for the different constructs of an Alloy model. The coverage provided by Test0 can be seen in Table I. AUnit coverage criteria look for the test suite's valuations to cover a range of shapes. For set based constructs (signatures (s), relations (r) and expressions (e)), the criteria relate to the size of the set produced by the valuation. For formula (f) based constructs, the criteria relate to the truth evaluation produced by the valuation.

B. Coverage-Based Test Suite

AUnit test cases can be automatically generated by incrementally producing valuations that satisfy new coverage requirements. To do this, one can append an Alloy formula that explicitly requires an uncovered requirement to be true. Then, by executing this formula, the SAT solver generates an instance, which we can use to start forming test case(s). The

command will be any paragraph in which the valuation covers a new criteria. Test generation terminates when all feasible criteria have been covered. For instance, we can generate a test to cover the requirement that a signature must evaluate to the empty set ('|s| = 0') for the signature Node. To do this, we would append '#Node = 0' to our model, execute the formula, and use the first instance found, the empty instance, as test's valuation. This would result in the following test case:

```
pred Val2 { no Node and no edges }
Test2: run { Val2[] and isDirectedTree[] }
```

The generated test case's command is isDirectedTree because: (1) the valuation is an instance of this paragraph, and (2) new criteria for isDirectedTree is covered by this valuation, e.g. "#iden = 0".

III. Extending the Grammar

When defining AUnit, our goal was to think conceptually about what would be a useful definition of a test case. When it came to adding a test case to a model, we looked to work with the existing supported grammar of Alloy. As a result, we would add a valuation as a named predicate paragraph in which the name had to start with "Val", and a command as an Alloy command in which the label had to start with "Test". However, now that we are looking to provide a comprehensive AUnit environment within the Analyzer, we can extend the Analyzer's grammar to provide a native, rather than ad-hoc, support. To achieve this, we have added three new extensions to the Analyzer's grammar: one to support declaring a test's valuations and two to support declaring a test's command.

A. Handling Valuations

To enable a user to easily specify a valuation, we have added a new paragraph type indicated by the keyword "val". A valuation paragraph does not allow for any parameter declarations, must be uniquely named, cannot have a return value, and consists of a block of constraints. There are no restrictions on the types of formulas which can appear in a valuation, enabling the user to specify complete as well as partial valuations. As an example, consider Test0 from section II. We would now declare our valuations as follows:

```
val SelfLoop {
  some disj N0, N1 : Node {
    Node = N0 + N1
    edges = N0->N0 + N0->N1 }
}
```

Aesthetically, there is not much difference between the two. However, there are several advantages to creating a dedicated paragraph type: the overall readability of the test case is improved, the user can provide better naming structures, e.g. "Val0" becomes "SelfLoop", and sanity checks can easily be supported within the Analyzer, e.g. ensuring a test case has referenced a valuation.

B. Handling Commands

To handle a test's command, we introduce two new attributes to Alloy. First, the "@Test" keyword denotes which Alloy commands are actually test cases. The "@Test" keyword must appear before any command label that is intended to be a test. For example, Test0's command would become:

```
@Test Test0: run {SelfLoop[] and !isDirectedTree[]}
```

The second, "@cmd:{...}", is a wrapper formula which allows for embedding a test case's command in the valuation paragraph. Any formulas within the curly brackets are marked as part of the AUnit test command. This functionality is particularly important in order to test predicates which have parameters. For example, consider adding the following paragraph to our model:

```
pred Reachable(start: one Node, end: one Node) {
   end in start.^edges
}
```

How would we write a test for the predicate? If we simply do the following:

```
@Test IsReachable:
run { SelfLoop[] and Reachable[N0, N0] }
```

The Alloy Analyzer will throw a compilation error, as there is no globally defined "N0" to be found. When passing parameters to a function call, the Analyzer needs to be able to discover the variables used. As a result, the command must be defined where the variables are defined, which means it must be declared in the 'val' paragraph structure. Therefore, we can use "@cmd" as follows:

```
val SelfReachable {
  some disj N0, N1 : Node {
    Node = N0 + N1
    edges = N0->N0 + N0->N1
    @cmd{ Reachable[N0, N0] } }
}
@Test IsReachable: run SelfReachable
```

The command of a test case can be any Alloy formula; therefore, we cannot just assume that any set equality ('=') declaration in a valuation paragraph outlines the valuations, and all other formulas are part of the command. By allowing the user to clearly express which formulas are part of the command, the introduction of the "@cmd" ensures that the user's true intentions are conveyed.

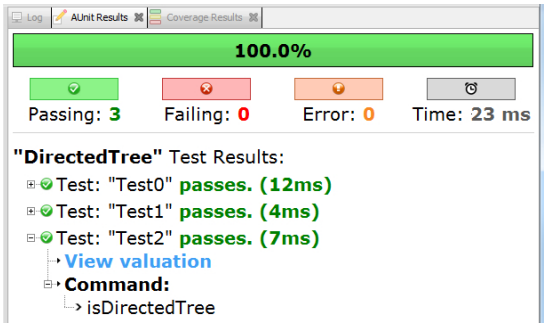
IV. AUnit Analyzer User Interface

The AUnit extension to the Alloy Analyzer is released as a stand alone jar – named the AUnit Analyzer – and is built on top of the existing Alloy Analyzer. All functionality provided in the latest stable release of the Analyzer is preserved. Our extension adds an additional usage scenario to the tool-set: AUnit testing. To see how this is achieved, we can look at how the new features can be applied together to test a model. In the Alloy Analyzer, the left pane is used to display the current Alloy model, while the right pane is used to display logging information. All AUnit-based results, except coverage-based highlighting, are designed to be displayed as different result tabs in the right pane. Throughout this section, we will be working with the directed tree model from section II and the tests we outline there (Test0, Test1, and Test2).

A. Executing a Test Suite

To run an AUnit test suite, one can either (a) select "Execute Test Suite" from the "AUnit" menu or (b) select the "AUnit"

Fig. 2: AUnit Test Execution Results



icon from the icon menu bar. Before executing any tests, the Analyzer will make a series of syntactic checks, such as ensuring that each test is uniquely named or that each test references *one* valuation. However, multiple tests can reference the same valuation. We do not enforce that a test case has an explicit command, in order to allow users to test just the facts of the model. Upon completion, the "AUnit Results" display tab will automatically populate the right pane. Figure 2 shows the "AUnit Results" report for our current test suite. The summary starts with a high level picture: how many tests passed/failed/produced an error, and the total execution time. Then, users can examine the details of each test, namely: the valuation, the command, the execution time, and the result. The "view valuation" action will produce a new pop-up window in which the user can examine the valuation textually or graphically. Since a fact formula must always be true, the command displayed will reflect both the user specified command, as well as any fact formulas in the model. By default, all passing tests are displayed with their information collapsed and all failing tests are displayed with their information expanded. To show the reporting structure, `Test2` has been expanded in Figure 2.

B. Calculating Coverage for a Test Suite

In the AUnit drop-down menu, there are two coverage options. First, one can toggle whether or not coverage is calculated for a test suite ("Calculate Coverage:"). Second, one can toggle whether or not to highlight the model according to the calculated coverage information ("Highlight Model:"). Figure 3 show the model coverage highlighting produced by our running example, in which green means covered, yellow means partially covered and red means not-covered. Valuation paragraphs and command paragraphs do not get highlighted.

After successfully executing the test suite, if "Calculate Coverage" is toggled to "Yes", the Analyzer will calculate model coverage for the entire model, regardless of which paragraphs were explicitly tested. Additionally, coverage is propagated: if a predicate invokes another paragraph, then the Analyzer checks to see what criteria are covered for the invoked paragraph. Figure 3 shows this information for our running example, which is displayed in the "Coverage Results" tab in the right pane. Coverage information is displayed by paragraph. For each paragraph, the coverage percentage is displayed on the right, and all of the constructs are listed below. Each construct shows its current coverage status (e.g. green dot means all the coverage criteria have been met) and each construct can be expanded to show the coverage status for all of its criteria, e.g. construct "`Node`" is expanded in Figure 3.

C. Coverage-based Test Generation

Prior work has shown $AGen_{Cov}$ to be efficient at producing small but robust test suites capable of detecting real-world faults [10]. Therefore, to help developers start integrating AUnit testing into their development practices, we wanted to support $AGen_{Cov}$ within our extension. To automatically generate test cases, the user can select (a) the "Generate Test Suite" option from the "AUnit" drop-down menu or (b) the "Test Gen" icon from the icon menu bar. Then, a pop-up window (Figure 4(a)) appears which asks the user to specify: (1) the model, (2) the paragraphs of interest (which signatures and predicates to generate tests over), and (3) the scope to be used. Once the test suite has been generated, the user is given the choice to label the expected behavior for the test cases. $AGen_{Cov}$ is an input generation technique: the valuations and commands pairs are automatically generated, but a human oracle is needed to say whether the behavior matches expectation or not. Figure 4(b) shows the interface that enables user to label test cases. The top pane displays the test case and asks the user if the valuation should be an instance of the listed command. To help the user properly label test cases, the bottom pane displays the valuation on the left and the model on the right. Users can view the valuation graphically, textually, and in a tree format.

D. Small Feature Changes

To bring AUnit to active Alloy users, building support in the Alloy Analyzer is the most direct path: the Analyzer is the official development environment for Alloy. In fact, one of the key strengths of Alloy as a language is the Alloy Analyzer itself. A common limitation of lightweight formal methods is often their lack of robust tool support compared to development environments that many programmers are familiar with, e.g. Eclipse or NetBeans for Java. The Analyzer provides a strong initial basis for Alloy development: a model editor interface with syntax highlighting, an execution environment, unsat-core highlighting, and the ability to inspect and evaluate instances from successfully executed commands.

However, for new users who are already struggling to learn the language, a development environment that lacks a handful of small low-level usage features can make the tool appear cumbersome, adding to the users frustration. Therefore, as we looked to incorporate AUnit, we also set out to provide a handful of small feature upgrades. First, we added a right-click menu for the model editor pane that supports many of the basic editing functions such as: copy, paste, undo, save, etc. Second, we added line numbers to the editor pane, enabling easier syntax debugging when a user get a compilation error. Lastly, we added close icons for tabs. The Analyzer does allow users

Fig. 3: AUnit Test Suite Coverage Results

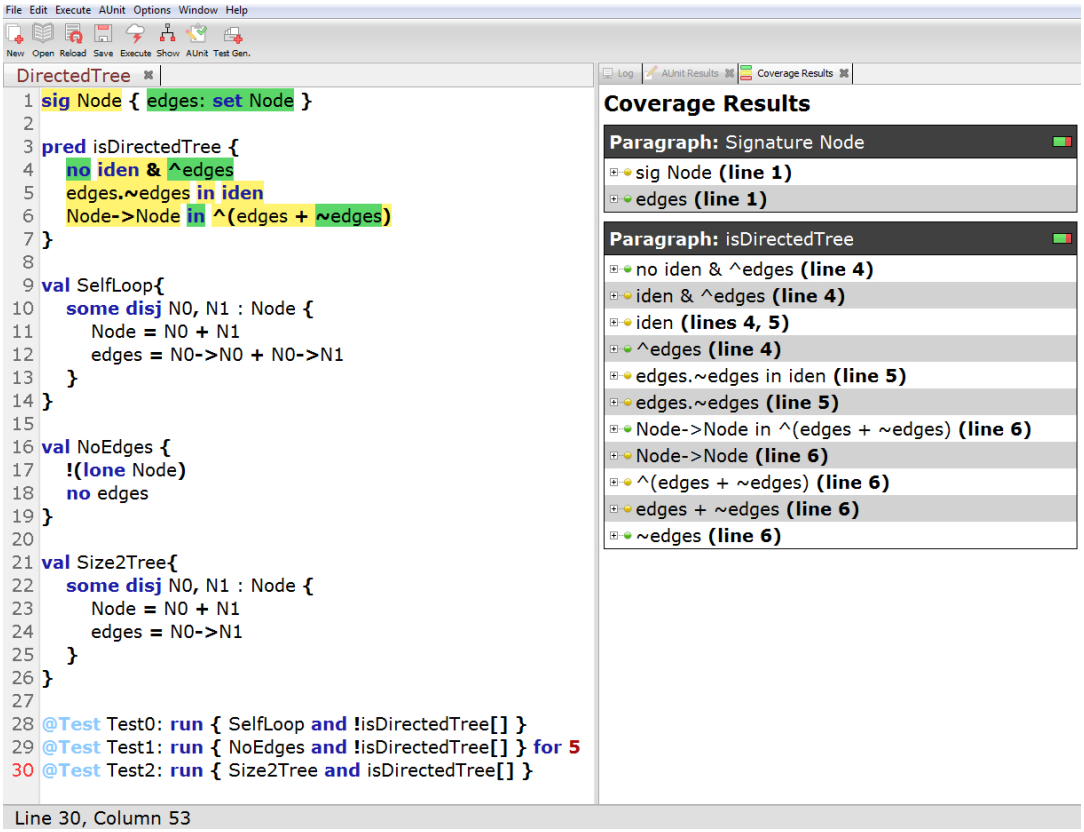
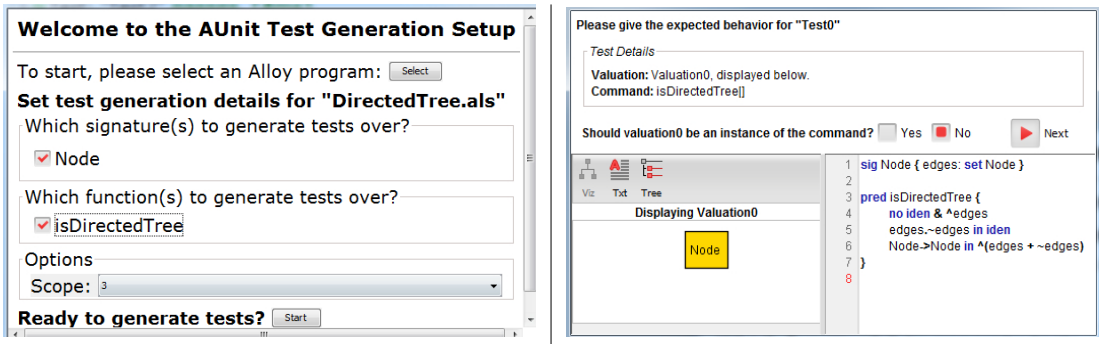


Fig. 4: $AGen_{Cov}$Test Generation GUI



(a) (b)

to have multiple models opened on separate tabs. However, in order to close the tab, one would have to have the tab actively displayed, go to the 'File' menu, and select 'Close.'

None of these features are necessary for the tool to function; however, they are small quality-of-life features that help smooth out a user's experience, especially if the user is already struggling with an unfamiliar language.

V. Lessons Learned

As we moved AUnit from the research world to the Alloy development world, we learned several lessons along the way. First, we discovered that we would have to merge our theoretical approach with the existing physical implementation. One big difference to reconcile was that the code-based implementation of Alloy does not view constraints in the same way that we did when we derived our coverage criteria. In AUnit, constraints which evaluate to sets are regarded as expressions and constraints which evaluate to true and false are regarded as formulas. We made this distinction because we naturally envisioned different coverage requirements for each. However, to the Alloy Analyzer, this distinction is irrelevant.

To account for this, we simply created our own abstract syntax tree which we use to find all coverage criteria for a model. Another common issue we ran into was that the current implementation of Alloy does not track as much information as we needed to fully support AUnit's coverage features. Therefore, we increased the information collected natively as the model is parsed.

Second, to design AUnit so it can in future handle most Alloy models, we considered support for the Alloy language at large. When we initially conceived of AUnit, we focused on a subset of the Alloy grammar centered around predicates and assertions. For instance, we did not think about abstract signatures or the impact of importing one model into another. To provide a seamless test environment, for all the supported functionality in the Analyzer, we had to ensure that AUnit either (1) supported the functionality or (2) ignored the functionality, allowing the user to still test all the AUnit supported aspects of the model. This included figuring our how AUnit tests should work for functions. Function paragraphs are different from predicates and assertions in that: (1) a function evaluates to a set and not a boolean value, and (2) is intended to return this set. Functions can be tested by specifying the AUnit command as follows: "`'func invocation' = 'anticipated return value'`". However, $AGen_{Cov}$ ignores functions for now.

Lastly, we had to learn how best to translate our research efforts into an intuitive and helpful user interface. This pushed us to view AUnit from the perspective of what an end-user would care about, separate from our focus and bias as creators of AUnit. For instance, while we had answered what an AUnit test case should be, we had to address the question: "If a test case fails, what information is useful to report?" This lead to a series of design decisions for the "AUnit Results" tab. First, we made the assumption that a user is more interested in digging through why a test failed than why a test passed. Therefore, the default display for a failing test is to start with its information expanded while a passing test starts with its information collapsed. Second, based on the assumption that the valuation is easier to debug graphically, we included a "view valuation" link so that a user can quickly inspect the valuation. As another example, when adding support for $AGen_{Cov}$, we had an opportunity to set up a user interface to facilitate the user providing the human oracle that the approach requires. This interface (Figure 4(b)) is not necessary to provide full technical support for $AGen_{Cov}$. However, as a user, this interface is very desirable.

VI. Conclusion and Future Work

Alloy is used by many developers to model system designs, and the Alloy Analyzer provides developers with a fully automatic analysis engine to explore their models. AUnit introduced the first framework to provide developers with a systematic approach to test their models. This tool paper introduces our efforts to bring AUnit into the Alloy development world by packaging AUnit as an extension to the Alloy Analyzer. Our initial extension introduces support for test creation, test execution, coverage calculations, and automated input generation. Our hope is that AUnit opens the door for the development of both new testing environments as well as development environments in future Alloy Analyzer releases, and that these front-end improvements helps further increase the usage of Alloy and its applications in developing more reliable software.

In future iterations of our AUnit extension, we envision four main feature additions. First, we want to provide support for the remainder of the Alloy grammar, namely by including a more direct test structure for function paragraphs. Second, we would like to provide support for users to input valuations graphically, rather than requiring users to provide the textual format. Third, we would like to build up more functionality for partial valuations. Currently, the SAT back-end is used to fill in the holes of a partial valuation, forming one complete valuation. However, a partial valuation typically represents multiple complete valuations. We would like to include an interface in which the user can iterate over the completed valuations found by the SAT solver, and incrementally extend the test suite and add to coverage as desired. Lastly, we would like to allow the user to choose the level of granularity and the type of coverage used to both calculate coverage and generate test suites.

Acknowledgment

We thank Jiaolong Yu, Darko Marinov, and Razieh Nokhbeh Zaeem for helpful comments and detailed discussions. This research was partially supported by the US National Science Foundation under Grant No. CCF-1718903.

References

[1] D. Jackson. Alloy: A lightweight object modelling notation. *ACM TOSEM*, 2002.
[2] D. Jackson. Alloy a language and tool for relational models. http://alloy.mit.edu/alloy/index.html, 2012.
[3] F. A. Maldonado-Lopez, J. Chavarriaga, and Y. Donoso. Detecting network policy conflicts using Alloy. In *ABZ*, 2014.
[4] S. Maoz, J. O. Ringert, and B. Rumpe. CD2Alloy: Class diagrams analysis using Alloy revisited. In *MODELS'11*, 2011.
[5] S. Maoz, J. O. Ringert, and B. Rumpe. CDDiff: Semantic differencing for class diagrams. In *ECOOP'11*, 2011.
[6] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *LISA'10*, 2010.
[7] N. Ruchansky and D. Proserpio. A (not) nice way to verify the openflow switch specification: Formal modelling of the openflow switch using Alloy. *SIGCOMM*, 2013.
[8] A. Sullivan. AUnit - a testing framework for Alloy. Master's thesis, University of Texas at Austin, 2014.
[9] A. Sullivan. *Automated Testing and Sketching of Alloy Models*. PhD thesis, University of Texas at Austin, 2017.
[10] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid. Automated test generation and mutation testing for Alloy. In *ICST*, 2017.
[11] A. Sullivan, R. N. Zaeem, S. Khurshid, and D. Marinov. Towards a test automation framework for Alloy. In *SPIN*, 2014.
[12] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.
[13] K. Wang. muAlloy – an automated mutation system for Alloy. Master's thesis, University of Texas at Austin, 2015.
[14] J. Yu. A prototype implementation of the AUnit test automation framework for Alloy. Master report, 2017.
[15] P. Zave. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.*, pages 49–57, 2012.