2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)

Crucible: Graphical Test Cases for Alloy Models

Adam G. Emerson
University of Texas at Arlington
Arlington, TX USA
adam.emerson@mavs.uta.edu

Allison Sullivan
University of Texas at Arlington
Arlington, TX USA
allison.sullivan@uta.edu

***Abstract*—Alloy is a declarative modeling language that is well suited for verifying system designs. Alloy models are automatically analyzed using the Analyzer, a toolset that helps the user understand their system by displaying the consequences of their properties, helping identify any missing or incorrect properties, and exploring the impact of modifications to those properties. To achieve this, the Analyzer invokes off-the-shelf SAT solvers to search for scenarios, which are assignments to the sets and relations of the model such that all executed formulas hold. To help write more accurate software models, Alloy has a unit testing framework, AUnit, which allows users to outline specific scenarios and check if those scenarios are correctly generated or prevented by their model. Unfortunately, AUnit currently only supports textual specifications of scenarios. This paper introduces Crucible, which allows users to graphically create AUnit test cases. In addition, Crucible provides automated guidance to users to ensure they are creating well structured, valuable test cases. As a result, Crucible eases the burden of adopting AUnit and brings AUnit test case creation more in line with how Alloy scenarios are commonly interacted with, which is graphically.**

***Index Terms*—Alloy, SAT Solver, Scenario Enumeration**

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 [13] is a relational modeling language. A key strength of Alloy is the ability to develop models in the Analyzer, an automatic analysis engine based on off-the-shelf SAT solvers, which the Analyzer uses to generate scenarios that highlight how the modeled properties either hold or are refuted, as desired. The user is able to iterate over these scenarios one by one, inspecting them for correctness. Alloy has been used to verify software system designs [35], [3], [32], [7], and to perform various forms of analyses over the corresponding implementation, including deep static checking [14], [10], systematic testing [18], data structure repair [34], automated debugging [11] and to synthesize security attacks [1], [20], [27].

However, to gain the many benefits that come from utilizing software models, the model itself needs to be correct. Unfortunately, while Alloy offers succinct formulation of complex properties, Alloy's support for expressive operators, such as transitive closure and quantified formulas, can make writing non-trivial properties challenging, especially for beginner users. In Alloy, there are two types of faults that can appear in a model: (1) *under-constrained* faults in which the model allows scenarios it should prevent, and (2) *over-constrained* faults in which the model prevents scenarios it should allow. To help detect these types of faults in an Alloy model, a unit testing framework, AUnit, was created [26], [25]. AUnit enables users to outline a specific scenario they expect their model to allow or prevent and then check that this behavior actually occurs. This improves upon the previous ad-hoc practices that require users to either (1) enumerate scenarios until finding one that is malformed or (2) enumerate all scenarios and realize one was missing, in order to determine if their model is faulty.

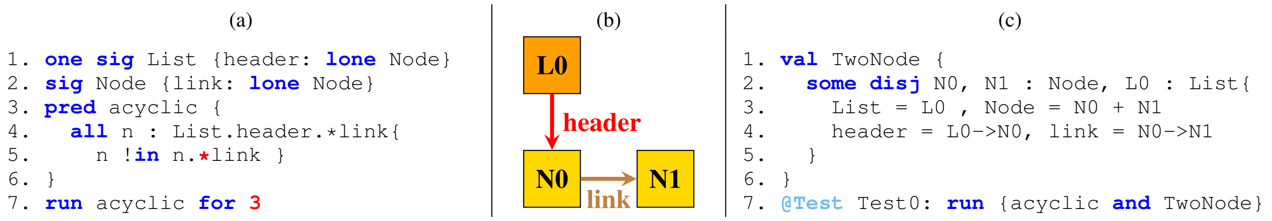
AUnit laid the foundation to bring a number of proven imperative testing practices to Alloy, including mutation testing [29], automated test generation [25], fault localization [30], automated repair [28] and partial synthesis of models [31]. These extensions help establish a comprehensive testing environment for Alloy that is similar to the robust testing support that imperative languages like Java have. Unfortunately, when it comes to the actual creation of an AUnit test case, the user is required to outline the valuation portion of a test case textually as a series of set equality statements that are wrapped around an existentially quantified formula.

However, valuations, which outline scenarios, are commonly interacted with graphically not textually. This results in a gap between the user's mental model of an Alloy valuation and the way the user currently has to create the valuation in order for the Analyzer to successfully reproduce it and perform unit testing. This paper addresses this issue by introducing *Crucible*, which establishes support for users to graphically create test cases. In addition, *Crucible* leverages the underlying model to help guide the user to create well-formed test cases by warning users when they attempt to create test cases that violate the model's structural constraints, which helps the user create stronger, more effective test suites.

In this paper, we make the following contributions:

Graphical Specification of Test Cases: We introduce *Crucible*, which enables users to create AUnit test cases graphically through a drag and drop interface.

2332-6549/23/$31.00 ©2023 IEEE
DOI 10.1109/ISSRE59848.2023.00065



(a)

```
1. one sig List {header: lone Node}
2. sig Node {link: lone Node}
3. pred acyclic {
4.   all n : List.header.*link{
5.     n !in n.*link }
6. }
7. run acyclic for 3
```

(b)

(c)

```
1. val TwoNode {
2.   some disj N0, N1 : Node, L0 : List{
3.     List = L0 , Node = N0 + N1
4.     header = L0->N0, link = N0->N1
5.   }
6. }
7. @Test Test0: run {acyclic and TwoNode}
```

Fig. 1. Faulty Alloy Model and Fault-Revealing AUnit Test Case

Automated Guidance: Based on the underlying model, *Crucible* will prevent users from creating malformed test cases and inform the user of which part of the model's signatures prevents the test case from being feasible.

Evaluation: We evaluate the overhead of *Crucible*'s translation between the graphical and textual representation of a test case. We also use key examples to highlight the need to support the graphical creation of test cases to ease the burden of creating test cases for models with complicated structures.

Open Source: We release *Crucible* as an open-source toolset at: https://github.com/Crucible-Alloy/Crucible.

II. Background

A. Alloy

To highlight how modeling in Alloy works, Figure 1 (a) depicts a faulty model of a singly-linked list with an acyclic constraint. Signature paragraphs and the relations declared within introduce atoms and their relationships (lines 1 - 2). Line 1 introduces a named set `List` and uses the relation `header` to express that each `List` atom has zero or one header node (`lone`). Similarly, line 2 introduces the named set `Node` and uses the `link` relation to express that each `Node` atom points to zero or one other node. Predicate paragraphs introduce named formulas that can be invoked elsewhere (lines 3 - 6). The predicate `Acyclic` uses universal quantification (`all`), set exclusion (`!in`), relation join (`.`), and reflexive transitive closure (`*`) to try to express the idea that "for all nodes in the list, no node is reachable from themselves following one or more traversals down the link relation." The fault, in red, can be corrected by replacing reflexive transitive closure ('`*`') with transitive closure ('`^`'), which will produce a set that will not include the node itself.

Commands indicate which formulas to invoke and what scope to explore. The command on line 7 asks the Analyzer to search for satisfying assignments to all the sets of the model (`List`, `header`, `Node`, and `link`) such that `Acyclic` is true using up to 3 `List` atoms and 3 `Node` atoms. A user can iterate over all the scenarios found by the SAT solver one by one. At a conceptual level, each scenario depicts behavior currently allowed by the modeled system. Figure 1 (b) graphically displays a scenario that a user would expect to be found by the Analyzer when the command at line 7 is executed: a list with two nodes and no cycles. Therefore, the user would expect to encounter this scenario at some point. However, due to the fault, this will never happen.

B. AUnit

AUnit addresses the need to have a systematic method to check the correctness of Alloy models [26]. Before AUnit, there was no formal notion of "testing" in the Analyzer. As a result, experienced users would employ a range of ad-hoc techniques, such as enumerating all scenarios – which can number in the thousands – and visually inspecting them for issues, a process which is both time consuming and error prone. Moreover, the number of scenarios can be in the thousands and the order scenarios are presented is based on the order the backend SAT solver finds them, which means the scenarios are effectively unordered. Altogether, this makes the enumeration-inspection process impractical. For instance, since scenarios are unordered, the user cannot count on inspecting them in order of increasing size, which would make it more feasible to catch a missing scenario.

The key insight behind AUnit is that unit testing, the most effective way to validate *code*, provides a blueprint on how to validate *models*. Specifically, an AUnit test case consists of two components: a *valuation*, which is an assignment to the sets and relations of the model, and a *command*, which specifies the Alloy formulas under test. A test case passes if the valuation is a valid scenario of the associated command; otherwise, the test fails. AUnit enables a user to directly ensure a specific scenario they have in mind is correctly generated – which checks for over-constrained faults – or prevented – which checks for under-constrained faults – without having to rely on encountering the scenario, or not, in the Analyzer.

To demonstrate, Figure 1 (b) and (c) depicts an AUnit test case graphically and textually which reveals the faulty behavior. This test case is based on a scenario which is valid for the correct model, but is incorrectly invalid for the faulty model. The valuation, outlined in lines 1-6 in Figure 1 (c), assigns all the sets (`List` and `Node`) and relations (`header` and `link`) of the singly-linked list to concrete values, creating a single scenario to reason over, by using existential quantification ('`some`') and the disjoint operator ('`disj`') to declare local variables and set equality ('`=`') to assign these local variables to constrain the sets of the model. The command given in line 7 in Figure 1 (c) outlines the `Acyclic` predicate as the formula under test. When the Analyzer executes the test case, the command is unexpectedly unsatisfiable, revealing the fault. Prior work has extended the Analyzer with the ability to natively declare AUnit test cases by extending the grammar with new keywords: `val` to outlined valuations and `@Test` to flag which Alloy commands refer to test executions [24].

C. Challenge: Specifying Test Cases Textually

The textual format, as seen in Figure 1 (c), can be tedious to provide, especially if the test case reasons over multiple states, contains numerous atoms or has higher arity relations. These features in a model can quickly bloat the length of the textual representation, impacting the readability, and thus usability, of an AUnit test. For instance, if a model has 10 signatures, then the atoms for all 10 signatures need to be accounted for as local variables declared by the existentially quantified formula. Then, the user needs to make a set equality formula for each signature. For a scope of 3, this would be declaring up to 30 variables and creating 10 separate set equality formulas for just the signatures. The test case would still need to create set equality formulas to account for any relations, which could utilize all 30 variables multiple times.

The textual representation is also not in line with the default way users inspect scenarios, which is graphically. As a result, requiring users to supply the textual representation increases the burden on the user to accurately translate their mental image of a scenario into a valid test case. In fact, recent work has demonstrated the importance of spatial cognition ability in solving Alloy tasks for both novice and expert users [17]. This is especially true for writing AUnit test cases, which requires the user to mentally picture a scenario of interest and then accurately write constraints to, in turn, generate that exact scenario. *Crucible* addresses these pain points by allowing the user to directly supply the graphical representation of a scenario, which *Crucible* will automatically translate to the textual representation.

III. *Crucible*

This section outlines important implementation details of *Crucible*, which is a standalone desktop application. We first present an overview of *Crucible*'s system architecture. Then we step over the process of creating test cases, including how *Crucible* ensures a user cannot create a malformed test case and how *Crucible* automatically translates graphical renderings to textual test cases.

A. Framework Overview

Figure 2 displays the high level software architecture for *Crucible*. *Crucible* looks to connect two main processes together: (1) the React graphical user interface (GUI), which the user will use to create AUnit test cases and (2) the Alloy Analyzer, which will execute the AUnit test cases. While the Analyzer is written in Java, our GUI is built using popular web technologies; Typescript, React, and Electron. To ensure the two processes are able to communicate, the Analyzer is wrapped in a SpringBoot REST API which is launched in conjunction with the GUI. This API handles the communication between processes when Alloy is needed, which occurs (1) when the model is initially parsed and its signature and predicate information is stored for use by *Crucible* and (2) when a test case is executed. Although it may have been a more obvious choice to write *Crucible* using a Java framework like JavaFX [21], which would do away with the additional overhead of an API, we opted for a web-based desktop platform for several reasons.

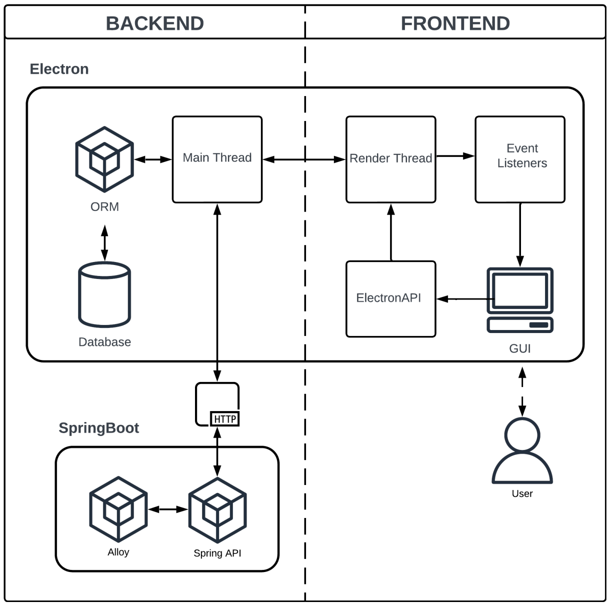


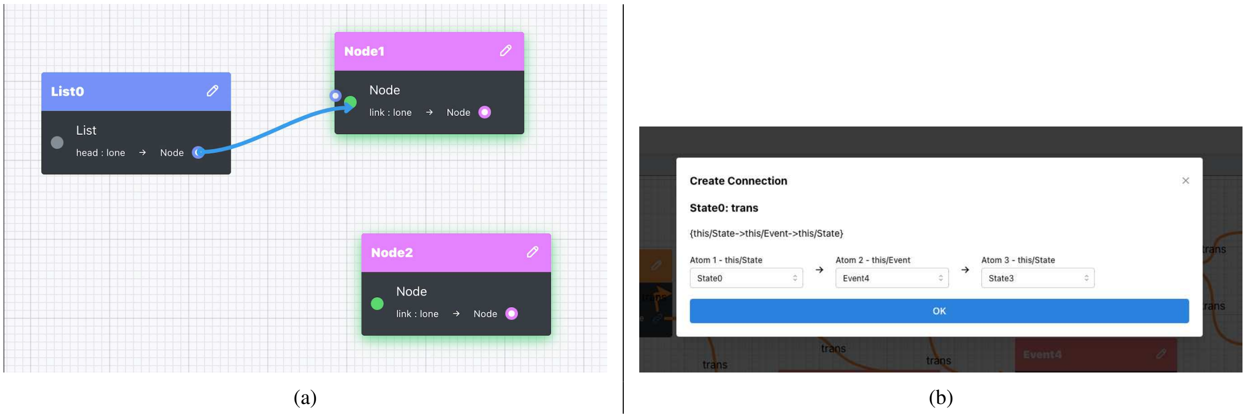
Fig. 2. *Crucible* Software Architecture Diagram

Firstly, the core purpose of *Crucible* is to improve both user experience and efficiency while writing AUnit test cases. With this in mind, it is critical that the application's interface is familiar, intuitive, and performant. Front-end frameworks like React, Angular, and Vue, have all been driving forces in UI development over the last decade [22]. These frameworks, particularly React, have grown to support a massive ecosystem of well-documented off-the-shelf primitive components that serve to expedite development of web and desktop applications alike. Thanks to widespread adoption of React and the GUI oriented nature of the web, the available open-source components are of a higher-quality than those which may be available within the Java ecosystem.

Secondly, of equal importance is the portability and accessibility of *Crucible*. By writing *Crucible* with web technologies, specifically React and Typescript, we ensure that deployment of *Crucible* is even more flexible than that afforded by Java. If in the future a third-party wished to host *Crucible* on a server for remote learning or some other application, it would be relatively trivial to port the codebase into a fully fledged web application. Notably, one of the main educational environments for Alloy is the Alloy4Fun website, which provides an online hosted platform for editing, sharing and interpreting Alloy models [16]. *Crucible*'s current form factor can more easily be integrated with Alloy4Fun than a JavaFX variant of *Crucible*. As a result, Alloy4Fun can feasibly be updated to give new users a native environment to easily address "how do I test my Alloy program?"

B. Creating a Project

To get started with *Crucible*, the user first needs to input their Alloy model into a new "Project". Each project consists of a single Alloy file, and acts as an organizational object



Create Connection

State0: trans

{this/State->this/Event->this/State}

Atom 1 - this/State: State0 → Atom 2 - this/Event: Event4 → Atom 3 - this/State: State3

OK

(a)

(b)

Fig. 3. *Crucible* canvas in action: (a) highlighting in green the allowed connections and (b) declaring higher arity connections.

for all of the tests written for the model. Upon selecting an Alloy file, the file path is sent via the API to the Java process, where it is passed into the Alloy Analyzer and converted into a JSON object of the model's signature and predicate definitions. The JSON object is then returned to the GUI process where it is cached in a SQLite database for subsequent usage. The SQLite database is used by *Crucible* to help provide automated guidance, as it contains all the information needed to know any structural constraints attached to a model's signatures, such as multiplicity constraints, and any defined relations. The SQLite object also stores all the information needed such that *Crucible* can easily allow users to test any predicate currently defined in the model, including ensuring that the user provides the required parameters for a predicate, if needed.

C. Creating a Test Case

Once the model has been imported and a project has been initialized, the user can then create any number of test cases. Each test is uniquely named, and consists of (1) a *canvas* onto which the user can spawn any number of atoms and connections, as allowed by the model, and (2) a set of all predicates and assertions declared in the model, by which the user can select which predicate(s) to consider under evaluation. Atoms are elements of signature sets and connections populate relations onto the canvas, e.g. to replicate the valuation in Figure 1 (b), `N0` would be a placeable `Node` atom and the red `header` directed line would be a placeable connection.

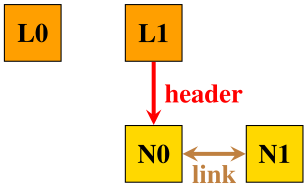
In *Crucible*'s project view, the model's signatures are presented as small tokens in a drawer menu on the left hand side. Each token displays the signature's name, relations, and multiplicity. Every token is assigned a color upon project initialization which can be edited to the user's liking. To build a test, atoms are dragged from the appropriate signature token and onto the canvas. Upon being added to the canvas, each atom is automatically given a unique nickname for use in command string generation and to help identify the atom as a predicate parameter. Once a sufficient number of atoms have been added to the canvas, connections can be made between them through a similar drag and drop interaction, or in the case of connections with a higher arity, a modal pop-up. Changes to a test case are saved automatically as they are made, further streamlining the process and allowing a user to focus on the task at hand.

D. Automated Guidance

As atoms are dropped onto the canvas, *Crucible* checks the current canvas state for multiplicity violations and alerts the user if they are attempting an addition that violates the model. Consider the `List` signature from our singly-linked list model:

```
one sig List {header: lone Node}
```

This signature uses the singleton multiplicity constraint, meaning that there can only be one `List` object for any valid scenario of this model. Therefore, if a user tries to form a test case with multiple `List` objects, *Crucible* will alert the user that she is attempting to violate the structural constraints of the `List` signature. This is an important detail for the user to be aware of, as any test with more than one `List` object will *always* be prevented by the model. Therefore, *Crucible*'s proactive guidance ensures the user will not incorrectly draw conclusions about the correctness of any predicates under test. To illustrate, if the user built the following valuation:



and checked that it is successfully prevented by the `acyclic` predicate, the user could build a false sense of security in the accuracy of their `acyclic` predicate, as the valuation will always be prevented due to the presence of two `List` atoms, regardless of any formulas in `acyclic`.

As a user initiates a connection interaction to add relations to the canvas, valid connection targets will be highlighted based on the defined relations in the model, as seen in Figure 3 where the two `Node` atoms are glow green to outline valid `header` connections. If a user attempts to make a connection to a non-valid target, they will be notified of the issue. In addition, if the user attempts to make a connection that violates a relational multiplicity constraint, they will again be notified and the action will be prevented.

To illustrate, consider the header relation:

```
1. one sig List {header: lone Node}
```

The header relation conveys two important pieces of information. First, the header relation is meant to connect a List atom to a Node atom. Second, the multiplicity constraint lone further restricts this by asserting that for each List atom, the header relation can only connect that List atom to either no Node atom or exactly one Node atom. As seen in Figure 3 (a), when a user wants to add a header relation to their test case, the user will see that the relation must start on a List object, and only end connections on Node atoms will be highlighted.

For higher arity (3+) relations, we do not currently enforce multiplicity constraints. However, if a higher arity connection is specified of the form "a->b->c" and the user deletes the connection "a->b," we automatically remove "b->c" from the canvas. In addition, to add a higher arity connection, we created a tailored modal that helps guide the user to specify each segment of the connection with drop-down menus that populate with only the valid atom options. Figure 3 (b) shows the higher arity relation modal. Since our singly linked list model only has binary relations, we use the LTS from our evaluation in Section IV-B to highlight this interface and its corresponding guidance.

Crucible's proactive nature of preventing users from creating test cases that violate the constraints outlined in signature paragraphs, and alerting users as to why what they are attempting to create is malformed, ensures that user is both aware of how the structural constraints of their model restrict the shape of valid valuations and ensures the user knows that the valuation is prevented because of these structural constraints, regardless of any command portion the user may have placed on the test case. This directly prevents the false sense of security that can be formed about a predicate mentioned earlier, where creating a test case with more than one list does not help us evaluate the acyclic predicate or *any* other system property the user writes.

As a tradeoff for the guidance we provide, users cannot directly form test cases for constraints enforced by the signature paragraphs of their model. We do believe that users should ensure their signature paragraphs are correct and modify them if they are not. Since *Crucible* proactively gives the user detailed error message pop-ups when the user tries to violate multiplicity constraints and grays out improper relation connections, the user still interactively explores these constraints within *Crucible*, enabling the user to still check the accuracy of their signature paragraphs, albeit indirectly. However, we feel the tradeoff is worthwhile to ensure users are actually testing the predicates they intend to.

E. Automated Translation

Running a test in *Crucible* is as simple as pressing a button. At runtime, the test's canvas is converted into an AUnit command string that the Alloy API can process and execute using the Analyzer. The command string is a series of valid Alloy formulas that, when executed, will produce just the scenario outlined on the canvas. To create the command string, *Crucible* processes each atom captured on the canvas, which includes tying the atom to its unique nickname and capturing all of the declared connections attached to this atom. Then, *Crucible* builds a mapping from each atom to the atom's associated signature. Once this mapping is formed, for each signature, *Crucible* generates an existentially quantified formula of the form:

```
some disj [nickname]* : [signature name] {
```

The disj keywords ensures that each variable name listed will produce a distinct atom for any satisfying instance. For example, in Figure 3 the following will get generated based on the state of the canvas:

```
some disj L0 : List {
some disj N0, N1 : Node {
```

where N0 and N1 cannot be represented by the same atom for any scenario produced by the Analyzer. *Crucible* processes each signature in the order they are declared in the model. Once all nicknames have been declared as local variables, *Crucible* generates a set equality formula of the form:

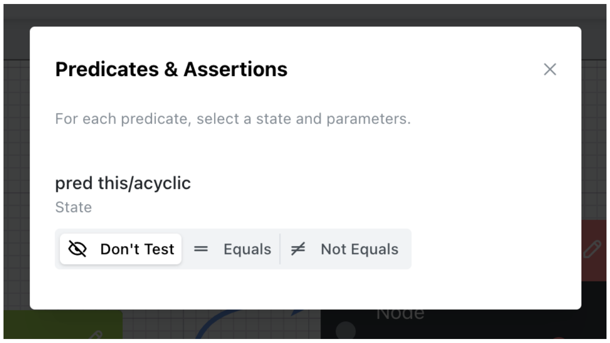
```
[signature name] = [nickname] (+ [nickname])* |
                   no [signaure name]
 [relation name] = [connection] (+ [connection])* |
                   no [relation name]
```

where (+) is set union. As a result, the value each signature set can take for any generated scenario is restricted to just the declared local variables of that type and nothing else. In addition, relations are restricted to the connections specified by the atoms. Likewise, the set equality formula must be declared within the scope of the local variables. If there are no atoms in the canvas for a signature or no connections for a relation, then the empty set operator (no) is used instead to ensure that this signature or relation does not appear in the corresponding scenario the Analyzer generates to satisfy the outlined valuation. For our example this will result in the following:

```
some disj L0 : List {  //Start of L0 scope
some disj N0, N1 : Node { //Start of N0, N1 scope
  List = L0,         Node = N0 + N1
  header = L0->N0, linke = N0-N1
```

Before running a test the user has the option of modifying the predicate(s) they wish to test. Users can adjust predicates by opening the predicate modal, where they will be able to assign atoms by nickname as parameters and chose one of the states for the predicate, as seen in Figure 4.

The states are "Don't Test" (null), where the predicate is not tested, "Valid", where the valuation is expected to be generated by the predicate, and "Invalid" where the valuation is expected to be prevented by the predicate. Based on the user's



Predicates & Assertions

For each predicate, select a state and parameters.

pred this/acyclic

State

Fig. 4. *Crucible* Predicates Modal

selection, *Crucible* will append the following information to the command string:

```
null: ""
true: [predicate name][(param)*]
false: ![predicate name][(param)*]
```

For our example, this will result in the following:

```
some disj L0 : List {  //Start of L0 scope
some disj N0, N1 : Node { //Start of N0, N1 scope
  List = L0,        Node = N0 + N1
  header = L0->N0,  linke = N0-N1
  acyclic[]
}}
```

Outlining the predicate under test within the scope of the local variables is important, as predicates can have parameters. For instance, if acyclic was defined as "`acyclic[l : List]`" then the predicate under test would become "`acyclic[L0]`," which the Analyzer will fail to compile if the predicate call is located outside the scope of the `L0` variable.

Once the command string is generated, the string is sent to the Java Process to be run in the Analyzer. If the command string is satisfiable, the API returns a success code to the GUI Process, and the user is notified that the test has passed. Alternatively, if the test is not satisfiable, the API returns the failure. It is worth noting that due to the multiplicity constraints being strictly enforced on the canvas, it is not possible to generate a failing test case without enabling one or more predicates, with the exception of higher arity relations which require further exploration.

IV. Evaluation

We evaluate *Crucible* in two ways. First, we evaluate the overhead of translating graphical renderings into executable test cases. Second, we conduct an illustrative case study over a select set of models to highlight how *Crucible* can ease the burden of creating test cases for models with tedious features for textual test case creation.

A. Overhead

Table I shows the runtime to translate the canvas state of *Crucible* into a command string for increasingly larger and larger test cases. Column **Model** conveys the model under evaluation. The next two columns outline the size of the model: column **#Sig** is the number of signatures and column **#Rel** is the number of relations in the model. To convey the size of the test case, column **#Atoms** displays the number of atoms on the canvas and column **#Con** shows the number of connections on the canvas. Column **Time[ms]** conveys the average time it takes (rounded to the nearest millisecond) across ten executions for *Crucible* to generate the command string once the execute test button in pressed. To perform the calculations, we create incrementally larger graphical test cases in *Crucible* for the two models we explore in our case study: the LTS model, which contains a higher arity relation, and the CV model, which contains a large number of signatures and relations.

TABLE I
Overhead of Translation

Model	#Sig	#Rel	#Atoms	#Con	Time[ms]
LTS	3	1	3	3	6
			6	6	9
			12	12	11
			24	24	14
			48	48	21
CV	5	4	9	20	8
			21	35	9
			36	60	11
			48	80	14
			60	100	25

The result of these benchmarks indicate that the conversion process of a canvas into a command string is negligible. Runtime appears to increase linearly as the number of atoms and connections does, but even with an impractically large model of 60 atoms and 100 arity-3 connections, the translation process on our modest workstation (a 2014 Macbook Pro) did not exceed an average of 25ms. With this is mind, we conclude that the graphical-to-textual translation process adds virtually no overhead when working on an Alloy model of a typical size, and is unlikely to be an issue for larger scale models.

B. Case Study: Debugging Real World Faulty Models

For our case study, we focus on two models from the Alloy4Fun benchmark [2]. Alloy4Fun is a online learning platform for Alloy whose exercises have users attempt to write predicates for various models, which are checked against a back-end oracle solution. Submissions to Alloy4Fun have been anonymized and made into an open source benchmark. These models represent faulty models created by new Alloy users. While AUnit is available for any Alloy user, we envision that new users are more likely to utilize AUnit. Our case study looks to highlight how different model structures can make writing AUnit test cases tedious and error prone.

1) Higher Arity Relations: Often times, Alloy models consist of binary relations (2-arity). For instance, in Figure 1, the relation `header` is a binary relation of the form `List×Node`. This is conceptually easier for a user to visualize mentally and put on paper, as the `header` can be envisioned a directed line that connects a single `List` atom to a single `Node` atom. However, in Alloy, it is possible for a relation of higher arity to be specified. To illustrate, consider the Labeled Transition System (LTS) model from the Alloy4Fun benchmark shown in Figure 5. Line 1 introduces the signature `State`, which

```
1. sig State { trans : Event -> State }
2. sig Init in State {}
3. sig Event {}
4.
5. //The LTS is deterministic.
6. pred inv3 {
7.   all s : State, e : Event | lone s.(e.trans)
8. }
```

Fig. 5. Faulty Model of a Labeled Transition System (LTS)

contains the relation trans. This relation is a tenary relation (3-arity) of the form State×Event×State. Rather than being a directed line between two atoms, trans indirectly connects two states through an intermediate Event atom. The idea of the trans relation is that the transitions between states are triggered by events; therefore, this intermediate Event atom is an important connection between the states.

For the remainder of the LTS model, line 2 introduces the Init signature as a subset (in) of the State signature, which conveys the initial state of the system, and line 3 introduces the signature Event that contains no relation itself. We elect to illustrate *Crucible*'s experience over predicate inv3, which is the third exercise in the LTS model on Alloy4Fun, as it involves the trans ternary relation in its formulation. The faulty predicate inv3 (lines 6 - 8) is meant to convey that the LTS is deterministic, meaning that for every state, every Event triggers either no transition or a unique transition to a next state. The faulty formulation uses an incorrect order of the relational joins. To illustrate, the following is the correct version of the predicate, with the difference highlighted in red:

```
all s : State, e : Event | lone e.(s.trans)
```

This error is a subtle change textually, but the fault results in a formula that is trivially *always* true. Namely, the faulty expression "e.trans" looks to form a relational join of the form Event with State×Event×State. Since there is a type mismatch, this first join will always produce an empty set. Since an empty set always satisfies the **lone** multiplicity constraint, this produces the trivially true behavior. To reveal this fault, the user needs an AUnit test case in which an Event triggers multiple possible state transitions for the same state.

Consider the following fault revealing test case, where the red text highlights the behavior the model should prevent:

```
some disj  State0, State1: State | Event0, Event1,
Event2: Event {{
State = State0 + State1
trans = State1->Event0->State0
      + State1->Event0->State1
Event = Event0 + Event1 + Event2
 Init = State1
}}
```

Since Event0 triggers two different transitions for State1, the valuation should not be generated, but the faulty predicate will produce it. For comparison, Figure 6 displays the same test case recreated in *Crucible*. For the visual test case, the user can see that State1 has two transitions, but both use Event0, as there are not connections drawn to Event1 or Event2. In both cases, a user is likely to spot the issue with the conflict being the only values population the trans relation.

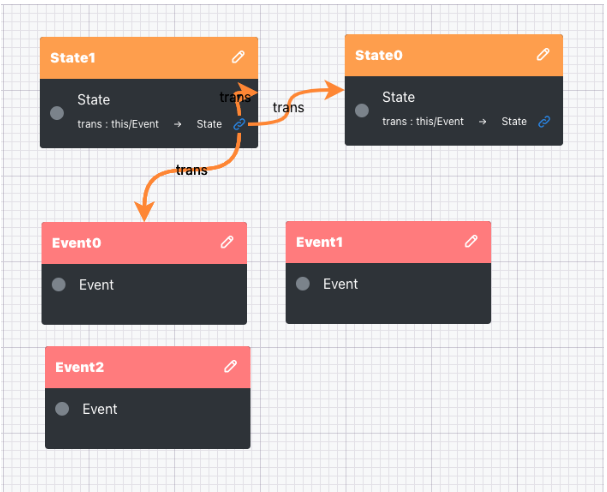


Fig. 6. The faulty LTS test case in *Crucible*

Consider the process of creating a larger test case. For example, the following test case that extends the previous one:

```
some disj  State0, State1, State2: State | Event0,
Event1, Event2: Event {{
State = State0 + State1 + State2
trans =
    State0->Event0->State0 + State0->Event0->State1
  + State0->Event0->State2 + State0->Event1->State0
  + State0->Event1->State1 + State0->Event1->State2
  + State0->Event2->State0 + State0->Event2->State1
  + State0->Event2->State2 + State1->Event0->State0
  + State1->Event0->State1 + State1->Event0->State2
  + State1->Event1->State0 + State1->Event1->State1
  + State1->Event1->State2 + State1->Event2->State0
  + State1->Event2->State1 + State1->Event2->State2
  + State2->Event0->State0 + State2->Event0->State1
  + State2->Event0->State2 + State2->Event1->State0
  + State2->Event1->State1 + State2->Event1->State2
  + State2->Event2->State0 + State2->Event2->State1
  + State2->Event2->State2
Event = Event0 + Event1 + Event2
 Init = State1
}}
```

which is significantly harder to follow textually. The extension to this test case is derived by using Amalgam [19] to create a maximal scenario based on the first test case. Using Amalgam allows us to highlight one of the largest (and therefore more complex) fault revealing test cases that a user could create with AUnit while remaining in the same scope as the small initial test case. There are several instances in which a user may be motivated to create larger test cases. For instance, if a user is looking to perform fault localization, repair or partial model synthesis with their AUnit test suite, past experiments reveal that larger test cases that encompass a wide degree of behavior result is notably better performance for these frameworks [30], [28], [31].

Realistically, if creating this test manually, the user is likely to copy, paste and then tweak assignments to the trans relation, which is an error prone process. In addition, if the user is textually specifying a relation this large, the user would realistically execute the test case and visually inspect the scenario the Analyzer produces multiple times as they build

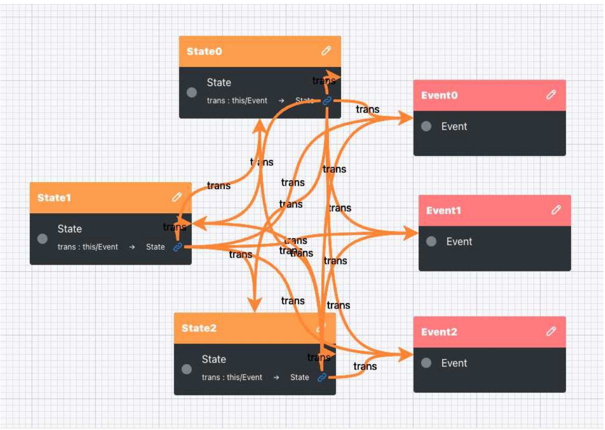

State0
State
trans : this/Event → State
State1
State
trans : this/Event → State
State2
State
trans : this/Event → State
Event0
Event
Event1
Event
Event2
Event
trans

Fig. 7. The maximal LTS test case in *Crucible*

up the valuation, to make sure the valuation actually matches their expectation.

In contrast, Figure 7 displays this same larger test case recreated in *Crucible*. Due to the different format of creating a test, *Crucible* removes the potential for copy-paste errors. More importantly, since *Crucible* provides a live graphical view as the user builds up a test case, *Crucible* removes the need to do repeated executions to spot-check the textual specification. These spot-checks do involve repeatedly running Alloy's backend SAT solver, although AUnit tests do not individually have a high overhead. While both the textual and graphical representations are cluttered, it is easier to implement lightweight interventions to make a graphical test case more readable. For instance, when a user hovers on a connection, we can gray out all unrelated atoms, easily bringing different portions of the `trans` relation into focus for the user. For now, users can drag and re-arrange the visual layout to better inspect connections post-creation. In contrast, there is no easy pathway to increase readability for the textual representation.

2) Numerous Signatures and Relations: In a recent profile of over 2000 different Alloy models [9], the median number of signatures and relations in a model is 8 and 2 respectively. Therefore, it is realistic to expect that a user may work with a model that contains a large number of signatures and relations. As the number of signatures and relations grow, the complexity of the valid scenarios for the models also grows. While Alloy defaults to a scope of 3 for commands, this scope is an upper bound of the size of *each* signature individually, and not a collective scope. As a result, if a user has 8 signatures in their model, a valid scenario can have up to 24 atoms. In addition, the scope does not place any restrictions on the size of relations. Therefore, these 24 atoms can be interconnected in 100s of ways. All of this increases the burden for a user to mentally visualize a scenario and then textually specify the corresponding test case. In fact, it would not be surprising in this instance, if the user first drew a scenario on paper before writing the corresponding test case.

To illustrate how *Crucible* can ease the burden of creating test cases with numerous signatures and relations, we select the `CV` model from the Alloy4Fun benchmark, which has 5 signatures and 4 relations, as seen in Figure 8. Line 1 introduces an abstract signature `Source`. As an abstract signature, `Source` cannot directly have atoms itself. The next two signatures extend the `Source` signature. Line 2 introduces the signature `User`, which contains two relations: `profile` connects a `User` to any number (`set`) of `Work` elements (line 3) and `visible` connects a `User` to any number (`set`) of `Work` elements as well (line 4). Line 6 introduces the signature `Institution` and line 8 introduces the signature `Id`, neither of which define any relations. Lastly, line 9 introduces the signature `Work`, which contains two relations: `ids` connects a `Work` atom to at least one (`some`) `Id` atom (line 10) and `source` connects a `Work` atom to exactly one (`one`) `Source` atom (line 11).

```
 1. abstract sig Source {}
 2. sig User extends Source {
 3.   profile : set Work,
 4.   visible : set Work
 5. }
 6. sig Institution extends Source {}
 7.
 8. sig Id {}
 9. sig Work {
10.     ids : some Id,
11.     source : one Source
12. }
13.
14. // The works publicly visible in a curriculum
15. // must be part of its profile
16. pred inv1 {
17.    User.visible in User.profile
18. }
```

Fig. 8. Faulty Model of a Curriculum Vitae Policy

The faulty predicate `inv1` (lines 16 - 18) attempts to use subset (`in`) to specify that any visible work is someone's CV must be part of that person's profile. The correct version of the predicate is:

```
all u:User | u.visible in u.profile
```

which is similar to the incorrect formula, but constrains the subset relationship to be true for each individual person (`u.visible`), rather than a universal perspective (`User.visible`). As a result, for the incorrect formula, a user could have a visible work in their CV as long as at least one person has that work in their profile, even if that person is not them. Consider the following fault revealing test case:

```
some disj  User0, User1: User | Work0, Work1, Work2:
Work | Id0 : Id {{{
   User = User0 + User1
profile = User1->Work0 + User1->Work1 + User1->Work2
visible = User0->Work0 + User0->Work1 + User0->Work2
     Id = Id0
   Work = Work0 + Work1 + Work2
    ids = Work0->Id0 + Work1->Id0 + Work2->Id0
 source = Work0->User1 + Work1->User1 + Work2->User0
no Institution
}}}
```

The issue with this test case is that `User0` is able to have `Work0` visible on their CV despite not having `Work0` in their profile because another user (`User1`) has the work in their profile. In fact, for this test case, this is true for every single work that is visible on `User0`'s CV. At first glance, this relationship may be easy to type and confirm textually. However consider the following test case, which is an extension of the previous test case:

```
some disj  User0, User1, User2: User | Work0, Work1,
Work2: Work | Id0, Id1, Id2: Id
{{{
   User = User0 + User1 + User2
profile = User0->Work1 + User0->Work2 + User1->Work0
        + User1->Work1 + User1->Work2 + User2->Work0
        + User2->Work1 + User2->Work2
visible = User0->Work0 + User0->Work1 + User0->Work2
        + User1->Work0 + User1->Work1 + User1->Work2
        + User2->Work0 + User2->Work1 + User2->Work2
    Id = Id0 + Id1 + Id2
   Work = Work0 + Work1 + Work2
    ids = Work0->Id0 + Work0->Id1 + Work0->Id2
        + Work1->Id0 + Work1->Id1 + Work1->Id2
        + Work2->Id0 + Work2->Id1 + Work2->Id2
 source = Work0->User2 + Work1->User2 + Work2->User1
no Institution
}}}
```

This test case is again derived by using Amalgam to create a maximal scenario based on the first test case [19] to also highlight one of the largest fault revealing AUnit test case a user could in theory create based on the scope. Again, the faulty behavior is revealed by the portion presented in red text. Figure 9 displays the same test case recreated in *Crucible*.

As with the LTS model, the main advantage of using *Crucible* is reducing the uncertainty of mentally re-creating such a long text chain for specifying a test case. In this case, rather than the majority of the complication being one relation, the complexity comes from the combination of different ways the atoms can relate to one another within the model. This still creates a high spatial cognitive burden to attempt to mentally visualize the test valuation from the text format, which is likely to result in the user incrementally writing the test and executing it to spot check that the test is written correctly. However, in both cases, there is a high spatial cognitive burden to attempt to mentally visualize the test valuation from the text format. While the large test case one again looks cluttered, the same lightweight visual interventions mentioned earlier apply here as well, while nothing can ease the text inspection burden.

V. Future Work

In *Crucible*'s current form, larger test cases can become quite cluttered, as is the case for Figure 7. Although this creates some overhead for the user as they are required to track connections visually, we hold that this overhead is less than that of the alternative – mentally visualizing a test case then writing a complex valuation textually. In future work, we will explore ways to reduce the visual clutter that a large test case creates by looking into new pathing techniques for connections and alternative visualization methods for canvases.

In Alloy, a user can customize the Analyzer's output with a robust theming subsystem. In *Crucible*'s current version, limited support for customization is available, with users having the ability to assign a custom color to each signature type. In a future release we aim to further enable the user to customize the appearance of their test with features such as the ability to rename atom instances, change the shape of the signatures, and highlight connections on hover. Enabling users to be more expressive when designing test cases will increase clarity and allow for the adoption of custom typologies within a user or organization's workflow.

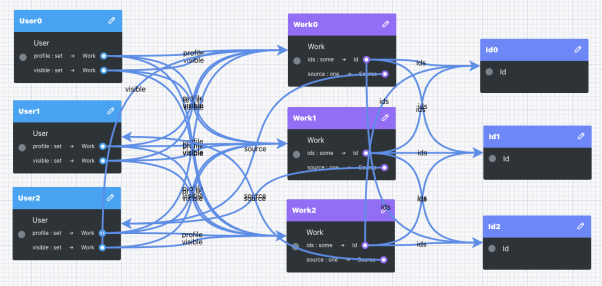


Fig. 9. The Curriculum Vitae Model test in *Crucible*

In addition, we hope to further improve our support for high arity (arity-3 and above) connections. This can be accomplished by improving the interface used to create arity-3 connections and by hardening our automated guidance techniques to ensure high arity relations have their multiplicities correctly enforced on the canvas. Optimized support for arity-3 and above connections will ensure that *Crucible* is useful for the majority of Alloy models in use today.

Finally, we plan to explore how to infer the underlying model structure from a collection of graphical test cases. This would alleviate the "how do I get started" burden of writing software models, which a recent user study found that both novice and expert Alloy users struggle to get started writing their model [17]. Specifically, based on an initial set of graphical test cases, we want to automatically create the signature paragraphs. To illustrate, from the test case in Figure 1 (b), we can conclude that there are two signatures (`Node` and `List`) and that there are two binary relations (`header` to type `List`×`Node` and `link` to type `Node`×`Node`). While a single test case does not let us confirm with absolute certainty the multiplicity of these relations, the user could supply additional tests that do. If not, we envision having an interactive process where we query the user for clarification.

VI. Related Work

Testing and Debugging Techniques for Alloy. Cruicble aims to ease the adoption of AUnit. There are a number of testing and debugging techniques which utilize AUnit tests: μAlloy is a mutation testing framework [25], AlloyFL is a hybrid fault localization technique that uses spectrum-based and mutation-based fault localization strategies to create a ranked list of suspicious locations [30], and ARepair is a generate-and-valid automated repair technique that uses AUnit test cases as an oracle to evaluate potential patches [28]. ICEBAR extends ARepair to consider built in Alloy assertions in addition to test cases to guide the repair [12].

There are also a number of repair techniques that use built in assertions in place of AUnit tests. ATR is an Alloy repair technique that tries to find patches based on a preset number of templates and uses Alloy assertions as an oracle [36]. BeAFix is an automated repair technique that uses a bounded exhaustive search [4]. TAR is a mutation-oriented repair technique that is aimed at repairing Alloy4Fun models, which are educational exercises [5]. FLACK is a fault localization technique that locates faults by using a partial max sat toolset to compare the difference between a satisfying instance of a predicate and a counterexample from an assertion over that predicate [37]. Alloy assertions can be used to check the accuracy of predicates, but assertions need to be written correctly themselves to be beneficial.

Drawing System Workflows. Our approach shares the spirit of storyboard programming, which uses user-provided graphical representations of data structures to synthesize code to perform data structure manipulations, based on the insight that it can be easier and more intuitive for a user to draw concrete data structure manipulations than to write the code [23]. Besides storyboard programming, there are other efforts related to drawing data structures and their transformations [8]. *Crucible* makes use of a similar insight: that it can be easier to draw examples of system behavior rather than to formally write the constraints. While not mathematical software models, there are several efforts to allow users to draw different UML diagrams [33], [6], [15]. These efforts, in particular FlexiSketch [33], allow users to free hand draw portions of UML diagrams. The lessons learned from the efforts helped informed our choice of where to draw the line between freehand drawings and a more structure drag-and-drop interface.

VII. Conclusion

AUnit test cases give users a simple and systematic way to spot check their Alloy models for correctness. In addition, a unit testing framework helps the model development process feel closer to that of writing imperative programs for novice software modelers. However, the need to specify AUnit test cases textually is a barrier to adoption for AUnit and its supported testing infrastructures, like fault localization and automated repair. By enabling users to build AUnit test cases graphically, we bring the creation of test cases more in line with how users interact with the output of Alloy models, which is largely a graphical process. *Crucible* takes this process a step further by helping guide users to create well-formed test cases based on the existing underlying model.

References

[1] Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: 2010 23rd IEEE Computer Security Foundations Symposium. pp. 290–304 (2010)

[2] Alloy4Fun Benchmark: https://zenodo.org/record/4676413 (2022)

[3] Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the Android permission system. Formal Asp. Comput. (2018)

[4] Brida, S.G., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.F.: Bounded exhaustive search of alloy specification repairs. In: ICSE (2021)

[5] Cerqueira, J., Cunha, A., Macedo, N.: Timely specification repair for alloy 6. In: Software Engineering and Formal Methods. pp. 288–303 (2022)

[6] Chen, Q., Grundy, J., Hosking, J.: An e-whiteboard application to support early design-stage sketching of uml diagrams. In: IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003. pp. 219–226. IEEE (2003)

[7] Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, Power, ARM, and C++. SIGPLAN Not. **53**(4), 211–225 (2018)

[8] Ding, C., Mateti, P.: A framework for the automated drawing of data structure diagrams. IEEE Transactions on Software Engineering **16**(5), 543–557 (1990)

[9] Eid, E., Day, N.A.: Static profiling alloy models. IEEE Transactions on Software Engineering pp. 1–1 (2022)

[10] Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: Efficient SAT-based bounded verification using symmetry breaking and tight bounds. TSE (2013)

[11] Gopinath, D., Malik, M.Z., Khurshid, S.: Specification-based program repair using SAT. In: TACAS. pp. 173–188 (2011)

[12] Gutiérrez Brida, S., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.: ICEBAR: Feedback-Driven Iterative Repair of Alloy Specifications (2023)

[13] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)

[14] Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: ISSTA (Aug 2000)

[15] Lank, E., Thorley, J., Chen, S., Blostein, D.: On-line recognition of uml diagrams. In: Proceedings of Sixth International Conference on Document Analysis and Recognition. pp. 356–360. IEEE (2001)

[16] Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., Silva, D.: Experiences on teaching alloy with an automated assessment platform. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods. pp. 61–77 (2020)

[17] Mansoor, N., Bagheri, H., Kang, E., Sharif., B.: An empirical study assessing software modeling in alloy. In: FormaliSE. p. To Appear (2023)

[18] Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: ASE (2001)

[19] Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": Enriching scenario exploration with provenance. In: FSE (2017)

[20] Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: LISA. pp. 1–8 (2010)

[21] OpenJFX: JavaFX - documentation (2023), https://openjfx.io/

[22] Saks, E.: JavaScript Frameworks: Angular vs React vs Vue. Master's thesis, University of Texas at Austin (2019)

[23] Singh, R., Solar-Lezama, A.: Synthesizing data structure manipulations from storyboards. In: FSE. pp. 289–299 (2011)

[24] Sullivan, A., Wang, K., Khurshid, S.: AUnit: A Test Automation Tool for Alloy. In: ICST DEMO Track. pp. 398–403 (2018)

[25] Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST (2017)

[26] Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a test automation framework for Alloy. In: SPIN. pp. 113–116 (2014)

[27] Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. IEEE Micro (2019)

[28] Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for Alloy. In: ASE (2018)

[29] Wang, K., Sullivan, A., Khurshid, S.: MuAlloy: A Mutation Testing Framework for Alloy. In: ICSE Demo Track. pp. 29–32 (2018)

[30] Wang, K., Sullivan, A., Khurshid, S.: Fault localization for declarative models in Alloy. In: ISSRE (2020)

[31] Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: ASketch: a sketching framework for Alloy. In: ABZ. pp. 121–136 (2018)

[32] Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: POPL (2017)

[33] Wüest, D., Seyff, N., Glinz, M.: Flexisketch: A mobile sketching tool for software modeling. In: International conference on mobile computing, applications, and services. pp. 225–244. Springer (2012)

[34] Zaeem, R.N., Khurshid, S.: Contract-based data structure repair using Alloy. In: ECOOP. pp. 577–598 (2010)

[35] Zave, P.: How to make Chord correct (using a stable base). CoRR **abs/1502.06461** (2015)

[36] Zheng, G., Nguyen, T., Brida, S.G., Regis, G., Aguirre, N., Frias, M.F., Bagheri, H.: Atr: Template-based repair for alloy specifications. In: ISSTA. p. 666–677 (2022)

[37] Zheng, G., Nguyen, T., Gutiérrez Brida, S., Regis, G., Frias, M.F., Aguirre, N., Bagheri, H.: Flack: Counterexample-guided fault localization for alloy models. In: ICSE. pp. 637–648 (2021)