HawkEye: User-Guided Enumeration of Scenarios

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

***Abstract*—Writing declarative models has numerous benefits, ranging from automated reasoning and correction of design-level properties before systems are built, to automated testing and debugging of their implementations after they are built. Alloy is a declarative modeling language that is well suited for verifying object-oriented designs. A key strength of Alloy is its scenario-finding toolset the Analyzer, which outputs all valid scenarios that adhere to the model's constraints up to a user-provided scope. However, in order for scenario-finding toolsets to be useful and not an undue burden, scenario-finding toolsets need to generate a relatively small but valuable collection of scenarios. This paper outlines Hawkeye, a novel interactive enumeration technique for the Analyzer that empowers the user to select which elements of a scenario the user wants to keep the same or differ in the next enumeration. Experimental results show that our technique can modify scenario enumeration without significant overhead on the size and complexity of the underlying SAT problem. Moreover, we highlight Hawkeye's ability to help users explore faulty models. Hawkeye is available at: https://github.com/alloy-hawkeye/Hawkeye.git**

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

I. Introduction

As software failures become increasingly costly, there is a growing need to leverage software models to improve the quality of software systems. Alloy is a first-order relational modeling language that is actively used in industry and research [18], [17], [27], [21], [10]. A key strength of Alloy is the ability to develop models in the Analyzer, a scenario enumeration toolset that lets users explore behavior enabled by their models. To achieve this, the Analyzer translates Alloy models into KodKod [33] formulas and invokes off-the-shelf Boolean satisfiability (SAT) solvers to search for scenarios, which are assignments to the sets of the model such that all executed formulas hold. As output, the Analyzer produces a collection of scenarios the user can iterate over.

While scenario enumeration is a highly useful application of formal methods from validating software designs [23], [31], [32], [25], [13] to program synthesis for security analysis of hardware [34], its effectiveness relies heavily on the quality and number of scenarios enumerated. In order to not hinder the applications which leverage these scenarios, scenario finding toolsets need to balance the number of the scenarios generated with the breadth of behavior covered by the scenarios. Creating too many similar scenarios can lead to verification technique that uses these scenarios to overlook important system functionality or for synthesis techniques to overfit to a narrow set of behavior. On the other hand, creating too many scenarios can significantly inflate the runtime of these techniques, often making the search space infeasible.

Alloy's enumeration is by design robust: Alloy produces a collection of scenarios that covers all valid behavior within a user provided scope. To help control the number of scenarios, during execution, the Analyzer appends symmetry breaking predicates to the execution, which remove many, but not always all, isomorphic scenarios. Despite using symmetry breaking constraints, the Analyzer frequently discover hundreds or even thousands of scenarios per executed command, which a user has to investigate one by one. Moreover, Alloy's discovery of scenarios is dictated by the order the SAT solver finds them, which can be random, can change between executions and can enumerate largely redundant scenarios back-to-back.

Specifically, in Alloy, scenario enumeration is provided by KodKod [33], which uses enumerating SAT solvers [8], [4], [28], [12], [16]. When the user desires another scenario, Kodkod follows the standard practice in modern SAT solvers for enumeration and adds a new clause to the propositional formula in conjunctive normal form (CNF) such that any new solution found to the SAT problem will differ from the previous solution for at least one primary boolean variable. While this enumeration strategy will avoid finding duplicate scenarios, it provides no assurances on how the next scenario will differ other than the fact that the scenario will be different. As a result, it is possible the next scenario enumerated could depict the smallest or largest change from the current scenario. All told, Alloy's default enumeration leaves a high burden on the user to try to find *valuable* scenarios.

In this paper, we present Hawkeye, a scenario enumerator for Alloy that looks to empower the user to help guide the enumeration process. Hawkeye is comprised of two key features. First, Hawkeye allows the user to interactively specify what elements of the current scenario the user wants to see stay the same in the next scenario. This allows the user to view scenarios that are extensions of or tangibly related to their current scenario. Second, Hawkeye allows the user to specify what in the current scenario should change for the next scenario enumerated. This functionality allows the user to guide the enumeration away from an unhelpful scenario and towards a potentially more valuable scenario. To give the user flexibility in expressing enumeration criteria, Hawkeye supports two levels of granularity: a high-level selection where users outline changes to make to the sets of the model and a low-level selection where users outlines changes to make to the individual atoms of the model. Additionally, Hawkeye allows users to mix both the level of granularity and type of enumeration constraints provided, empowering the user to fine tune which scenario is enumerated next.

In this paper, we make the following contributions:

- **User-Directed Enumeration**. We present a technique that empowers users to direct scenario enumeration by allowing users to specify what elements of the current scenario they want to preserve or differ.
- **Open Source**. We release our enumeration framework as an open-source extension of the latest stable release of the Analyzer. This allows users to gradually explore our new enumeration technique while still adhering to the original workflow of the Analyzer. Hawkeye can be found at https://github.com/alloy-hawkeye/Hawkeye.git.
- **Evaluation**. We present an experimental evaluation using a variety of subject models used to evaluate recent advancements to Alloy. We evaluate the overhead of Hawkeye's enumeration and the impact different types of user specified constraints have on the search space.
- **Case Study**. We explore how Hawkeye can help users find faults in real world faulty models in comparison to using Alloy's default enumeration.

II. Worked Example

This section presents a small but representative example of an Alloy model to introduce some key concepts of Alloy, the Analyzer's enumeration, and our technique Hawkeye.

A. Alloy and the Analyzer

To highlight how modeling in Alloy works, Figure 1 (a) depicts a model of a singly-linked list. Signature paragraphs and the relations declared within introduce atoms and their relationships (lines 1 - 2). Line 1 introduces `List` as a named set of atoms and declares `header` as a binary relation that conveys the idea that each `List` atom points to zero or one ('`lone`') header node. Line 2 introduces `Node` as a named set of atoms and defines the binary relation `link`, which asserts that every node will either point to nothing or point to a subsequent node. Predicate paragraphs introduce named first order logic formulas that can be invoked elsewhere (lines 3 - 6). The `Acyclic` predicate uses disjunction ('`or`') and existential quantification ('`some`') to express the idea "either the list empty or some node is the list terminates the list." Commands indicate which formulas to invoke and and upper bound on what scope to explore (line 7). The command in Figure 1 (a) instructs the Analyzer to search for a scenario using exactly 1 `List` atom and at most 3 `Node` atoms such that the `Acyclic` predicate evaluates to true.

B. Default Scenario Enumeration

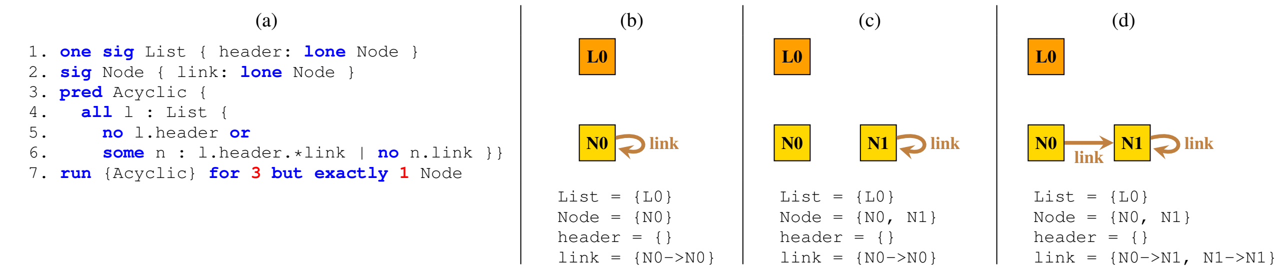
Figure 1 (b), (c) and (d) displays the first three scenarios found by the Analyzer in both the graphical and textual formats available to the user. Scenarios in Alloy are created by making assignments to the sets in the model, where each signature and relation is represented as their own set. In this paper, we refer to the elements within a set as *atoms*. For example, in Figure 1 (b), `List` is a set and `L0` is a list atom. A user can step through all the scenarios found by the SAT solver, inspecting them for correctness.

In Alloy, when the user desires another scenario, Alloy's backend KodKod adds a new clause to the propositional formula such that any new solution will differ from the prior solutions. While this encoding practice prevents duplicate scenarios, it does not guarantee that the next scenario enumerated will be in any way related to or markedly different than the current scenario. For instance, the three scenarios in Figure 1 all represent an empty list. The difference in behavior that these scenarios depict all involve the various behavior allowed by nodes disconnected from the list, whose acyclicity we do not care about. As a collection, these scenarios are not likely to be of high value to the user. However, in the Analyzer, there is no way for a user to express that they now want to see behavior for nodes within the list. Instead, a user has to exit the enumeration, append constraints to the model that force there to be at least one node in a list, and re-execute the command.

C. User-Guided Scenario Enumeration

Hawkeye empowers the users to express elements of the current scenario that they want to see stay the same or change. In particular, Hawkeye allows users to select specific sets, which represent the different signatures and relations, and/or specific atoms, which are the individual elements populating the sets, to keep the same or differ. To illustrate how Hawkeye works and its benefits, we highlight how a series of actions in Hawkeye can help the user explore the solution space for the singly linked list model in Figure 1. For the scenario in Figure 1 (b), we can guide Hawkeye to generate a more valuable scenario by actively telling Hawkeye to populate the list. To do this, the user can select to change the relation `header`, which will force `header` to not be empty, meaning there will be at least one node in the list. This selection results in the scenario in Figure 2 (a). From there, we can instruct Hawkeye to keep the `header` relation but generate a scenario with a different `link` to target exploring a larger list. This produces the scenario in Figure 2 (b), which captures an unexpected behavior allowed by our model where a node not in the list (`N1`) can point to a node in the list (`N0`). This behavior is now revealed early to the user, and the user can evaluate if this should be valid or not.

To continue to explore the search space, we can repeatedly ask Hawkeye to keep the sets `header` and `Node` the same. This results in the scenarios displayed in Figure 2 (c), (d) and (e). If the user tries to apply the same criteria again to the scenario in Figure 2 (e), she will be told that there are no more scenarios. Therefore, after enumerating these five scenarios, the user now knows that she has investigated all possible behaviors allowed by her model for a list with a header node and two node atoms. In contrast, using Alloy's default enumeration, the scenarios in Figure 2 appear as the 7th, 10th, 9th, 4th, and 12th scenarios respectively. While this collection of scenarios is not too far spread out, the scenarios in between all vary from scenarios with 1 node to scenarios



```
one sig List { header: lone Node }
sig Node { link: lone Node }
pred Acyclic {
  all l : List {
    no l.header or
    some n : l.header.*link | no n.link }}
run {Acyclic} for 3 but exactly 1 Node
```

(b)

```
List = {L0}
Node = {N0}
header = {}
link = {N0->N0}
```

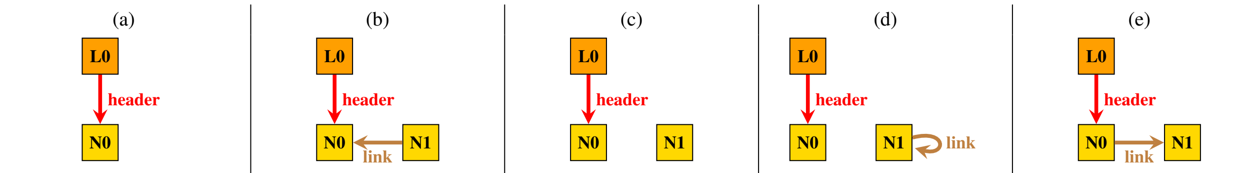
(c)

```
List = {L0}
Node = {N0, N1}
header = {}
link = {N0->N0}
```

(d)

```
List = {L0}
Node = {N0, N1}
header = {}
link = {N0->N1, N1->N1}
```

Fig. 1: Alloy model of a singly-linked list and a corresponding scenario.



(a) Generated by asking Hawkeye to change Figure 1 (b)'s `header`.

(b) Generated by asking Hawkeye to keep Figure 2 (a)'s `header` but change Figure 2 (a) `link`.

(c) Generated by asking Hawkeye to keep Figure 2 (b)'s `header` and `Node`.

(d) Generated by asking Hawkeye to keep Figure 2 (c)'s `header` and `Node`.

(e) Generated by asking Hawkeye to keep Figure 2 (d)'s `header` and `Node`.

Fig. 2: User-Guided Enumeration using Hawkeye.

with 3 nodes, making the connection between the scenarios hard to piece together. Additionally, after enumerating Figure 2 (e) as the 12th scenario, the user still has to enumerate 38 additional scenarios before confirming that the scenarios in Figure 2 (b) - (e) are the only valid scenarios for a list with a header node and two node atoms.

Hawkeye is designed to allow users to interact with the enumeration engine in order to streamline the generation of valuable scenarios. As illustrated, a user can go from a scenario that is not interesting to them, such as Figure 1 (b) where the list is empty, to a more useful scenario, such as Figure 2 (a), by using Hawkeye to guarantee that the next scenario enumerated will populate the list. Furthermore, by subsequently utilizing Hawkeye to outline portions of the scenario to retain, users can explore closely related scenarios back-to-back instead of using the SAT solver's discovery order. This allows a user to spot interesting behavior in the current scenario and then immediately choose to enumerate all the other scenarios under which that same behavior can arise.

III. Technique

In this section, we describe Hawkeye's encoding to enforce the user's guidance (Section III-A) and how we apply this encoded during the enumeration (Section III-B).

A. Encoding

In Alloy, scenarios are produced by KodKod, which generates a DIMACS formatted CNF formula equivalent to the invoked constraints and uses a back-end SAT solver to find a satisfying solution. KodKod then translates this solution into a scenario. Specifically, KodKod creates a list of all possible atom assignments that can populate a set and ties those to unique integer values, e.g. $\{$`L0->N0`$, 1\}$ for the `header` relation. These integers are the primary variables that make up the CNF formula passed to the SAT solver. While solving the CNF formula, the SAT solver determines if each primary variable should be positive (true) or negative (false). A positive assignment means the atom is in the scenario, e.g. "1" means `L0->N0` is in the `header` relation while "-1" means `L0->N0` is not in `header`. To enumerate a new scenario, KodKod instructs the SAT solver to find another satisfying assignment to the variables with an additional CNF clause that asserts that any new assignment found must differ from all previously discovered assignments by at least one primary variable.

Hawkeye augments Alloy's default enumeration by creating new CNF clauses that encode the user's guidance and must also be satisfied for the next solution found. Hawkeye supports two different encodings. First, to keep elements of the scenario constant, Hawkeye appends a CNF clause that requires the *relevant* primary variables to retain the same truth value for the next solution. Users can elect to keep specific atoms the same or to keep entire signatures and relations the same. Atoms are individually represented by a primary variable v. Hawkeye appends a CNF clause that asserts v if v is true for the current scenario and $-v$ if v is false, forcing v to stay the same truth value. Signatures and relations are represented by a collection of primary variables. In this case, Hawkeye appends a sequence of CNF clauses that asserts each primary variable mapped to the targeted signature or relation stays the same. To illustrate, consider the scenario in Figure 2 (b) and the following mapping of atoms to integers for the `header` relation: $\{$`L0->N0`$, 1\}$, $\{$`L0->N1`$, 2\}$ and $\{$`L0->N2`$, 3\}$. Adding the CNF clause $\{1\}$ keeps `N0` as `L0`'s header while adding the clauses $\{1\}, \{-2\}, \{-3\}$ keeps the entire `header` relation the same for the next scenario enumerated.

Second, to force elements of the scenario to change, Hawkeye appends a CNF clause that asserts the next scenario must

Algorithm 1: User-directed scenario enumeration.

```
Input: Kodkod solver solver, Previous scenarios prev, Set of
       relations sameSet, Set of relations diffSet, Set of
       atoms sameAtoms, Set of atoms diffAtoms.
Output: The next scenario with respect to the user-provided
        constraints on consistency and variance.
1  newClauses = { }   // set of new clauses for users choice
2  sameVars = {sameAtoms}   // set of unique ids for variables
3  foreach $\rho \in$ sameSet do
4      sameVars = sameVars $\cup$ primaryVariables($\rho$)
5  diffVars = { }   // map of rels to unique ids for variables
6  foreach $\rho \in$ diffSet do
7      diffVars.get($\rho$).add(primaryVariables($\rho$))
8  // Add user specified constraints - same, diff-atoms
9  for $i \leftarrow 1$ to solver.numPrimaryVars do
10     if $i \in$ sameVars then
11         newClauses.add(solver.valueOf($i$))
12     if $i \in$ diffAtoms then
13         newClauses.add(solver.valueOf($i$) ? -$i$ : $i$)
14 // Add user specified constraints - diff-sets
15 foreach $\rho \in$ diffVars do
16     c = new int[diffVars.get($\rho$).size()]
17     for $i \leftarrow 1$ to solver.numPrimaryVars do
18         if $i \in$ diffVars.get($\rho$) then
19             c[j] = solver.valueOf($i$) ? -$i$ : $i$
20             j++
21     newClauses.add(c)
22 solver.cnfClear(); // Clear out the cnf clauses stored
23 // Add cnf clauses to prevent duplicates
24 for $i \leftarrow 1$ to prev.size() do
25     solver.addClause(prev.get(i))
26 // Add cnf clauses for the users constraints
27 for $i \leftarrow 1$ to newClauses.size() do
28     solver.addClause(newClauses.get(i))
29 solution = solver.solve()
30 if solution == null then return   // no (new) solution found
31 // generate clause to prevent duplicate of new sol.
32 neg = new int[numPrimaryVars.size()]
33 int j = 0
34 for $i \leftarrow 1$ to solver.numPrimaryVars do
35     neg[j] = solution.valueOf($i$) ? -$i$ : $i$
36     j++
37 prev.add(neg)
38 output(solution)
```

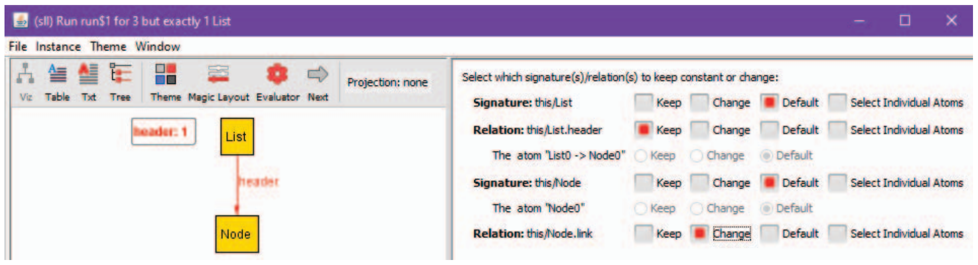
differ by at least one of the *relevant* primary variables. Again, users can assert differences at an individual atom level or at a signature and relation level. To encode changing an atom, Hawkeye appends a CNF clause that asserts $-v$ if v is true for the current scenario and v if v is false, flipping v's truth value. For example, adding the CNF clause $\{-1\}$ ensures the scenario after Figure 2 (b) will not have `N0` as `L0`'s header. For each targeted signature or relation, Hawkeye appends a clause that requires the solutions to differ with respect to at least one of the primary variables that correspond to that signature or relation. This "at least one" encoding follows the same format as traditional enumeration; however, it is expressed over a restricted subset of the primary variables instead of across all primary variables. If multiple signatures and relations are targeted, then Hawkeye appends a separate "at least one" clause for each. Importantly, appending a separate clause ensures that *each* selected set changes. To illustrate, the clause $\{-1\ 2\ 3\}$, which is interpreted as "-1 or 2 or 3," is used to ensure the next `header` assignment differs from Figure 2 (b).

B. User-Directed Enumeration

Algorithm 1 depicts Hawkeye's user-directed enumeration strategy. Hawkeye's enumeration algorithm's takes four different inputs to capture the user's guidance: (1) `sameSet` which are sets in the model the user wants to hold constant, (2) `diffSet` which are sets in the model the user wants to change, (3) `sameAtoms` which are atoms in the scenario the user wants to hold constant and (4) `diffAtoms` which are the atoms in the scenario the user wants to change. Users can provide all these different inputs for any given enumeration and is not restricted to one. Hawkeye's enumeration algorithm also takes two inputs that reflect the current enumeration problem: (1) `solver`, which is the KodKod instance depicting the current enumeration and (2) `prev`, which is a set of CNF clauses that are used to guarantee a new solution found by the SAT solver differs from any previous solution.

To start, Hawkeye builds a collection of primary variables that need to be kept constant (`sameVars`) and a collection of primary variables that need to change (`diffVars`) based on the user's input. If the user specified an atom, then Hawkeye's internal mapping has already connected the given atom to its primary variable (line 2). However, if a user selects a set, Hawkeye first collects all primary variables that represent the set. For sets that need to change, we use a map to store this information so we can appropriately append a separate CNF clause for each set. With the relevant primary variables collected, lines 9 - 13 and lines 15 - 21 implement the encoding outlined in Section III-A to enforce the user's guidance. If the user does not provide any guidance information, then no additional clauses are generated and Hawkeye's enumeration algorithm simply performs Alloy's default enumeration.

The remainder of the algorithm generates (1) the new scenario and (2) a new clause to ensure this scenario is not found again. On line 22, we clear out the current CNF clauses. By clearing out the CNF clauses, Hawkeye discards the user-specified requirements from the previous enumeration, ensuring that guidance is provide scenario-by-scenario and not applied across the entire enumeration. Then, we add back clauses to define the new enumeration. Specifically, we append (1) the CNF clauses the encode that new solution must differ from all previous solutions, which are stored in `prev` (lines 24 - 25) and (2) the CNF clauses that encoded the user's guidance (lines 27 - 28). If the solver successfully finds a new solution, we generate the CNF clause needed to ensure any future solution varies from the newly discovered one and append this clause to `prev` (lines 32 - 37). Lastly, the new scenario is displayed to the user or the user is informed that there are now more scenarios.


(sll) Run run$1 for 3 but exactly 1 List
File Instance Theme Window
Viz Table Txt Tree Theme Magic Layout Evaluator Next
Projection: none
header: 1
List
header
Node

Select which signature(s)/relation(s) to keep constant or change:

Signature: this/List	Keep	Change	[✓ Default]	Select Individual Atoms
Relation: this/List.header	[✓ Keep]	Change	Default	Select Individual Atoms
The atom "List0 -> Node0"	Keep	Change	[✓ Default]	
Signature: this/Node	Keep	Change	[✓ Default]	Select Individual Atoms
The atom "Node0"	Keep	Change	[✓ Default]	
Relation: this/Node.link	Keep	[✓ Change]	Default	Select Individual Atoms

Fig. 3: Hawkeye User Interface

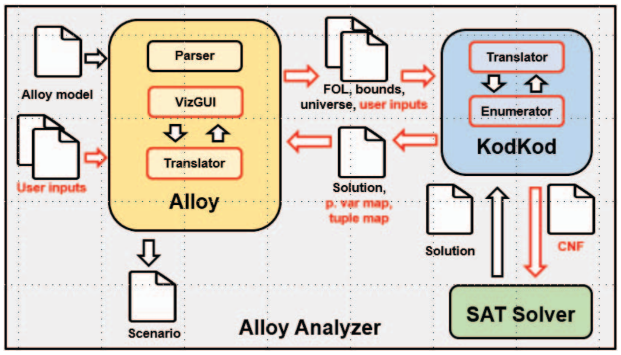


Fig. 4: Hawkeye Framework Overview

IV. Implementation

Hawkeye is implemented as an extension to the Analyzer v.5.0.1, the latest stable release of the Analyzer [1]. Importantly, since Hawkeye extends the main IDE for Alloy, users can maintain their current workflow while slowly exploring the new functionality Hawkeye provides.

Hawkeye extends the Analyzer's enumerator interface, VizGUI, as depicted in Figure 3 to collect the user's guidance. This interface is intended to be robust as it allows users to mix and match both the granularity and the type of enumeration change in one location. In particular, for each signature in the model, Hawkeye's interfaces lists the name of the signature and every atom of that type present in the scenario. These atoms are presented as an indented list and are unselectable until the user selects "Select Individual Atoms" at the signature level. This design choice is intended to prevent malicious input, as a user can either select an action for an entire signature or for individual atoms of that signature but not both. In addition, if a signature is abstract, the user is notified that they are not allowed to provide constraints for the signature. After displaying a signature and all of its atoms, Hawkeye's interface lists all of the relations defined by that signature and gives the same high-level or low-level selection options. Once all the information related to one signature has been displayed, the next signature is listed. The order in which signatures and relations are displayed is intended to be easy to follow, as the order replicates the order of declaration in the model.

To process the user's input, Hawkeye augments the enumeration workflow in the Analyzer, as seen in red in Figure 4. Hawkeye extends the communication pipeline between VizGUI and the different translation processes that occur during enumeration. Specifically, during execution, an Alloy model first gets translated into a KodKod model which then gets translated into a boolean logic formula in CNF format. Information is stored in one direct for this process; however, Hawkeye needs to seamlessly translate artifacts from Alloy to their CNF primary variables and from the primary variables back to Alloy. To that end, Hawkeye builds and maintains a mapping between the primary variables that make up the CNF encoding and their corresponding KodKod tuples and from the KodKod tuples to the Alloy atoms they depict. This mapping information is utilize to streamline Hawkeye's performance and interfaces. For instance, when a user selects an atom to reason over, Hawkeye can quickly access the atom's primary variable mapping to generate an appropriate CNF clause. Additionally, Hawkeye uses the mapping between KodKod tuples and Alloy atoms to populate the list of atoms to display for the user interface. Hawkeye also augments KodKod's enumerator to implement Algorithm 1. As output, the user is either presented with a new scenario that adheres to their preferences in the Analyzer's enumerator interface, or if the call to the SAT solver was unsatisfiable, the user is informed that no such scenario exist.

V. Evaluation

We evaluate Hawkeye over a benchmark of Alloy models collected from recent advancements made to Alloy [22], [35]. The models are a combination of models from the Analyzer's example set (**addr**, **gene**), models of protocols (**abc**, **bempl**, **grade**), models of data structures (**arr**, **btree**, **ctree**, **dll**, **graph fsm**, **sll**) and models of logic puzzles (**nqueens**). All experiments are performed on Windows 10 with 2.4GHz Intel Xeon CPU and 16 GB memory.

In this section, we address the following research questions:

- **RQ1:** What is the overhead of Hawkeye's enumeration strategy?
- **RQ2:** What impact do the different types of enumeration requests have on the valid scenario space?

A. Set Up

1) Baseline: We collect an initial set of results using the default enumeration strategy of the Analyzer, which we consider as a baseline for scenario enumeration performance.

TABLE I: Baseline results from the Analyzer.

Model	#PVrs	#Cls$_s$	#Cls$_e$	#Scr	T$_{avg}$	T$_{tot}$	Scp
addr	45	1551	1718	167	0.93	161	3
arr	67	1210	4981	3771	0.48	1855	3
bempl	38	500	1533	1033	0.42	424	3
btree	140	3318	4135	817	1.10	915	3
cd	29	3749	4013	264	3.06	824	5
ctree	40	1308	1780	472	0.92	440	5
digraph	20	202	6545	6343	0.34	2239	3
dll	72	4530	5227	679	0.85	601	3
fsm	40	1499	8020	6521	0.26	1734	5
gene	86	4474	4538	64	2.66	174	6
grade	48	604	3730	3126	0.33	1074	3
nqueens	330	34670	34728	58	35.37	2098	10
sll	15	302	352	50	0.75	44	3

To collect the results, we automatically enumerate all solutions in the Analyzer and add in a monitor to collect performance metrics related to size and time. To establish our baseline, Table I depicts the Analyzer's default enumeration performance. Column **Model** reflects the name of the model under evaluation. Columns **#PVrs**, **#Cls**$_s$, **#Cls**$_e$ show the number of primary variables, the number of clauses used to find the first solution, and the number of clauses used to find the last solution respectively. These columns together convey the size of the problem passed to the SAT solver. To capture attributes of the search, column **#Scr** depicts the number of scenarios found, column **T**$_{avg}$ shows the average time to find a scenario, column **T**$_{tot}$ depicts the total time to find all scenarios and column **Scp** is the scope. All times displayed in all tables are in milliseconds. Colums **#Cls**$_s$ and **#Cls**$_e$ are presented separately so that we can see how the number of clauses scales as we enumerate scenarios. In the Analyzer's standard enumeration, the number of clauses grows linearly with the number of scenarios found, as one clause is always to prevent finding a duplicate of the current solution. The number of primary variables and scope do not change across all experiments, so they are presented once in Table I and not repeated.

2) Configurations: Our experiments explore four different configurations of Hawkeye:

- **same-set:** We randomly select a set from the model to keep the same throughout the whole enumeration.
- **diff-set:** We re-use the set from the 'same-set' configuration but force the set to change throughout the whole enumeration.
- **same-atom:** We randomly select an atom from the first scenario enumerated to hold constant throughout the whole enumeration.
- **diff-atom:** Throughout the whole enumeration, we randomly select an atom from the current scenario to change for the next scenario.

As with our baseline, we automatically enumerate all solutions and collect various performance metrics. Importantly, these configurations are set up to evaluate the overhead of the different encodings Hawkeye provides, which capture all the different ways users can give guidance. The results for the four different configurations can be seen in the following tables: Table II, which shows Hawkeye's performance for the same-set configuration, Table III, which shows Hawkeye's performance for the diff-set configuration, Table IV, which shows Hawkeye's performance for the same-atom configuration, and Table V, which shows Hawkeye's performance for the diff-atom configuration. For Tables II and III, column **Set** communicates which set of the model was held constant across the enumeration. Similarly, for Table IV, column **Atom** displays the atom that was held constant.

B. RQ1: Overhead

We asses the overhead of Hawkeye's enumeration strategy by considering two metrics. First, we look at how Hawkeye's encoding increases the size of the underlying SAT problem. Second, we look at the impact Hawkeye's changes to the SAT problem have on the runtime.

1) Size of SAT Problem: For the underlying SAT problem, the number of primary variables does not change as we search for new scenarios but the number of clauses does change. Based on Hawkeye's design, which does not retain clauses generated to meet the user's guidance, Hawkeye should not negatively impact how the number of CNF clauses grows. This is supported by comparing the increase in the number of clauses, captured by **#Cls**$_s$ and **#Cls**$_e$, to the number of scenarios found, captured by **#Scr**. For Tables III, IV, V, we expect Hawkeye's encoding under these configurations to add one permanent clause to prevent duplicates and one temporary clause to capture the user's guidance, which **#Cls**$_e$ confirms. For Table II, Hawkeye's encoding will add an additional clause for every primary variable tied to the set being held constant. This results in a larger difference between the growth in clauses and the number of scenarios found. However, this amount is not significant compared to the total number of clauses and by design, these clauses are not preserved.

2) Complexity of SAT Problem: While the size of the SAT problem is not greatly impacted, as long as a user adds guidance, Hawkeye's encoding does add further constraints that any new scenario must now also satisfy, which restricts the valid space for the next scenario and should increase the search time. To measure the impact this can have on the complexity of the SAT problem, we measure the time it takes to find a new scenario, referred to as the average solve time, captured by column **T**$_{avg}$.

As seen in Tables II, III, IV, and V, the average solve time increases for all configurations compared to the baseline, as expected. Usually, this increase in average solve time is nominal, with most average solve times increasing by less than 10 milliseconds. The largest increase in average solve time occurred for the **nqueen**'s model across all four configurations. This increase ranges from 113.03 milliseconds, a 3.2x increase, under the 'same-set' configuration to 349.13 milliseconds, a 9.9x increase, under the 'same-atom' configuration. Even then, the user is not likely to notice a 300 milliseconds delay to view a scenario. As a result, when a user gives guidance using Hawkeye's interface, she will often not notice an impact in the time Hawkeye takes to display the next scenario.

TABLE II: Hawkeye result for the ‘same-set’ config.

Model	#Cls$_s$	#Cls$_e$	#Scr	T$_{avg}$	T$_{tot}$	Set
addr	1551	1618	40	2.68	312	listed
arr	1210	1460	247	2.31	1408	Element
bempl	500	607	104	1.58	486	Researcher
btree	3318	3951	624	21.05	15531	left
cd	3749	4013	264	9.12	3343	Class
ctree	1308	1421	103	2.74	664	color
digraph	202	216	10	1.63	64	Node
dll	4530	4659	120	5.81	1232	nxt
fsm	1499	1580	76	2.19	457	State
gene	4474	4520	10	10.81	278	spouse
grade	604	1191	578	1.70	1989	asc_with
nqueens	34670	34684	4	148.4	965	Queen
sll	302	315	10	2.72	103	Node

TABLE III: Hawkeye results for the ‘diff-set’ config.

Model	#Cls$_s$	#Cls$_e$	#Scr	T$_{avg}$	T$_{tot}$	Set
addr	1551	1719	167	2.21	931	listed
arr	1210	4369	3158	9.60	39313	Element
bempl	500	1513	1012	2.83	4162	Researcher
btree	3318	3684	365	10.53	5468	left
cd	3749	3751	1	5.5	29	Class
ctree	1308	1781	472	2.92	2412	color
digraph	202	482	279	0.87	611	Node
dll	4530	4905	374	11.78	5559	nxt
fsm	1499	1652	152	1.92	812	State
gene	4474	4537	62	10.20	1448	spouse
grade	604	3684	3079	7.66	30821	asc_with
nqueens	34670	34699	28	156.31	5442	Queen
sll	302	324	21	1.50	134	Node

TABLE IV: Hawkeye results for the ‘same-atom’ config.

Model	#Cls$_s$	#Cls$_e$	#Scr	T$_{avg}$	T$_{tot}$	Atom
addr	1551	1674	122	3.24	887	Name1
arr	1210	2639	1428	9.54	17357	>5
bempl	500	742	241	2.14	1155	Researcher0->Key0
btree	3318	4119	800	34.80	31898	Node1
cd	3749	3940	191	8.73	2640	Class1
ctree	1308	1607	298	4.19	2229	Node2->Node1
digraph	202	2488	2285	5.96	15812	Node0->Node1
dll	4530	5208	678	22.9	16706	Node0
fsm	1499	5410	3910	14.94	68999	State1->State2
gene	4474	4513	38	12.31	987	Man0->Eve0
grade	604	2044	1439	5.74	11184	Class2->Student0
nqueens	34670	34715	44	384.5	18992	Queen4->5
sll	302	349	46	1.7	322	Node1

TABLE V: Hawkeye results for the ‘diff-atom’ config.

Model	#Cls$_s$	#Cls$_e$	#Scr	T$_{avg}$	T$_{tot}$
addr	1551	1573	21	2.96	208
arr	1210	1270	59	2.35	453
bempl	500	555	54	1.53	337
btree	3318	3325	6	4.0	156
cd	3749	3754	4	9.2	113
ctree	1308	1313	4	4.0	57
digraph	202	204	1	2.5	11
dll	4530	4535	4	6.4	165
fsm	1499	1501	1	5.5	23
gene	4474	4476	1	22.0	82
grade	604	654	49	1.74	372
nqueens	34670	34675	4	171.4	1101
sll	302	308	5	1.67	34

However, the increase in average solve time can be noticeable when we consider the impact an increase in average solve time can have on the total runtime. Hawkeye often produces a longer total runtime (**T**$_{tot}$) than the baseline despite Hawkeye enumerating less scenarios, as seen in all configurations except ‘diff-atom.’ For example, the **fsm** model in the ‘same-atom’ configuration saw the largest magnitude increase in average solve time at a 56.5x increase, which resulted in a 39.8x increase in total runtime. As a result, Hawkeye takes one minute longer than the baseline to enumerate 2611 less scenarios. The tradeoff being that Hawkeye exchanges runtime for the knowledge that the 3,910 scenarios Hawkeye enumerates in this configuration show all the different behavior allowed by a specific atom.

In general, adding clauses to guide enumeration restricts the number of valid scenarios for that next enumeration. As a result, Hawkeye often increases the time to find a new scenario, which, in turn, increases the time to enumerate all scenarios. However, given Hawkeye’s interactive nature, the user will likely not notice the impact when enumerating a single scenario. Additionally, while the total runtime is impacted, all experiments finish in under 2 minutes, which is still fast and such a small overhead should not deter users from adopting Hawkeye.

C. *RQ2: Valid Scenario Space*

We expect the type and level of change the user is looking to make with respect to the current scenario to reduces the number of possible valid scenarios to be found. To measure this impact, we look at the total number of scenarios enumerated.

When we compare the number of scenarios found from the ‘same-set’ and ‘diff-set’ configurations, which reason over an identical set, we can see that the ‘same-set’ configuration frequently finds less scenarios. Conceptually, when a user asks to keep a portion of a scenario the same, the user is significantly restricting the space of valid scenarios because only scenarios that extend the selected behavior can be generated. Meanwhile, if the user asks to change a portion of a scenario, the user has restricted the space of valid scenarios, but the restriction expresses that the scenario should take any variety of other behavior instead. This latter restriction is usually not as limited of a perspective as the prior restriction.

This relationship is inverted at the atom level. Notably, guidance related to atoms that should differ can result in the user making a choice that severely limits the number of possible scenarios, including the possibility of directly violating constraints of the model, which results in no new scenario. As Table V shows, the configuration ‘diff-atom’ often results in a small number of scenarios. In this configuration, the models with low scenario counts have multiplicity constraints that restrict the size of some signatures. The early termination of these models occurred when the atom selected to change violated a multiplicity constraint, resulting in an unsatisfiable call. In contrast, guidance related to atoms that should stay constant may not noticeably reduce the valid scenario space. For instance, under the ‘same-atom’ configuration, **dll** enumerates one less scenario than the baseline because the selected atom, `Node0`, is in every scenario except one.

In general, asking Hawkeye to keep a set the same often reduces the number of valid scenarios more than changing a set. Meanwhile, asking Hawkeye to keep an atom constant may not impact the possible next scenario significantly while asking to change a specific atom can lead to an early termination.

D. Threats to Validity

Our evaluation is designed to evaluate the overhead of Hawkeye's different encoding which capture all the different unique ways the user can influence enumeration. As a result, our evaluation does not capture the interactive nature of Hawkeye in three main ways. First, the times reported are just the time to find the solutions, but Hawkeye does expect the user to inspect a scenario and decide on any enumeration constraints to apply for the next scenario. Second, a user may ask Hawkeye to generate a new scenario with a mix of both granularity levels and types of enumeration constraints. Third, our configurations were held constant over an entire enumeration; however, this is not required by Hawkeye and a user my change their guidance form one enumeration to the next. Therefore, our results may not generalize to the way users choose to interact with Hawkeye.

VI. Case Study

Hawkeye is designed with the idea that by giving users an active roll in the enumeration and not relying simply on the order of discovery by the SAT solvers, users will engage more with Alloy's enumerator as they benefit from exploring more valuable scenarios. This is motivated by Alloy's incremental development life-cycle where a user writes a constraint, executes a command to exercise the constraint, enumerates a few scenarios to make sure the constraint seems correct and then the user repeats this process with a new constraint. When checking for correctness, Alloy user can find two basic kinds of faults: (1) overconstraint, where the formula rules out valuations that the user wanted to allow and (2) underconstraint, where the formula allows valuations that the user wanted to rule out.

Alloy's default enumeration makes detecting faults difficult: the user either needs to enumerate all scenarios, which can number in the thousands, to notice one is missing or the user needs to randomly enumerates an unexpected scenario. Additionally, since Alloy's default enumeration can sometimes be very random and other times very redundant, users can be deterred from exploring enough scenarios to adequately have confidence in the correctness of their models. In this section, we explore how users can interact with Hawkeye to reveal faults in real world faulty models. Moreover, we show how the interactive nature of Hawkeye's enumeration can help users diagnose the fault. We focus on faults written by new Alloy users who are likely to benefit from Hawkeye's ability to allow users to actively check for and explore specific behavior, something new users lament about learning Alloy [5].

1) Overconstrained Faults: Overconstrained faults are often revealed when a user enumerates scenarios and fails to see a particular scenario. Using Alloy's default enumeration, we can see from Table I that this might mean the user needs to look at just 50 scenarios or as many as 6,000 scenarios. Using Hawkeye, the user is likely to investigate significantly less scenarios to discover the absence of a scenario. To illustrate, considering the following real-world overconstrained model:

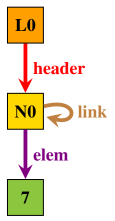
```
sig List { header: lone Node }

sig Node {
  link: lone Node,
  elem: one Int }

pred Sorted(l: List) {
  all n: l.header.*link | n.elem <= n.link.elem }
run {Sorted[List]} for 3 but exactly 1 List
```

This model is from a graduate student assignment [36]. The predicate `Sorted` is supposed to outline that for all nodes in the list, the nodes appear in ascending order based on their `elem` value. The user accidentally restricted the valid behavior of `Sorted`. Specifically, the subformula in the quantified formula requires every node in the list to have a populated `link` relation in order for the formula to be true. As a result, the Analyzer will never generate a list that contains a `Node` atom with an empty `link` relation, even though a node without a link can still appear in a valid sorted lists.

The first scenario enumerated by Hawkeye is the following:



The user can immediately use this scenario to reveal the overconstrained fault. Based on this scenario, the user may want to quickly confirm that the `link` does not need to be present. If the user asks Hawkeye to keep `header`, `Node` and `elem` the same while changing `link`, then the user will then be told there are no valid scenarios. Since the user targeted a specific change and discovered it was infeasible, the user gains the additional information that their constraints seem to prevent nodes from having an empty `link` relation. Using Alloy's default enumeration, there are 113,188 scenarios for this command. The user will need to inspect enough of these scenarios to feel confident that the scenarios being generated never allow a node to have an empty link relation. In the worst case, the user will inspect all of them.

Due to the `elem` relation, the number of scenarios generated for this model is particularly large. The scope for integers in Alloy is a bit-width, meaning that for a scope of 3, a node's `elem` can range from -8 to 7. This results is a lot of redundancy in scenarios. For instance, for the scenario outlined above, Alloy will enumerate 15 lists that have the same structure as this scenario but each scenario will have a different value populating the `elem` relation. While these scenarios contain the same high-level structure, these scenarios are non-isomorphic with respect to the identity of atoms (i.e. `elem=N0->7`, `elem=N0->6`). Therefore, Alloy's default symmetry breaking will preserve all 15. Yet, exploring all 15 of these scenarios may not be of high value to the user depending on the constraints being evaluated. Using Hawkeye,

(a)

```
1. one sig FSM { start: set State, stop: set State }
2. sig State { tsn: set State }
3. fact OneStartAndStop {
4.    some s1: State | s1 in FSM.start
5.    all s1, s2: State {
6.       s1 = FSM.start and s2 = FSM.start => s1 = s2 }
7.    some s1: State | s1 in FSM.stop
8.    all s1, s2: State {
9.       s1 = FSM.stop and s2 = FSM.stop => s1 = s2 }
10. }
11. run {} for 5
```

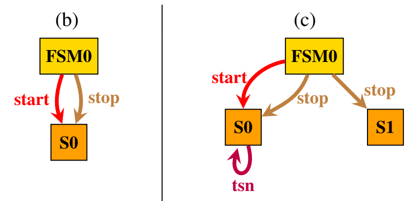


Fig. 5: Revealing Underconstrained Faults Using Hawkeye.

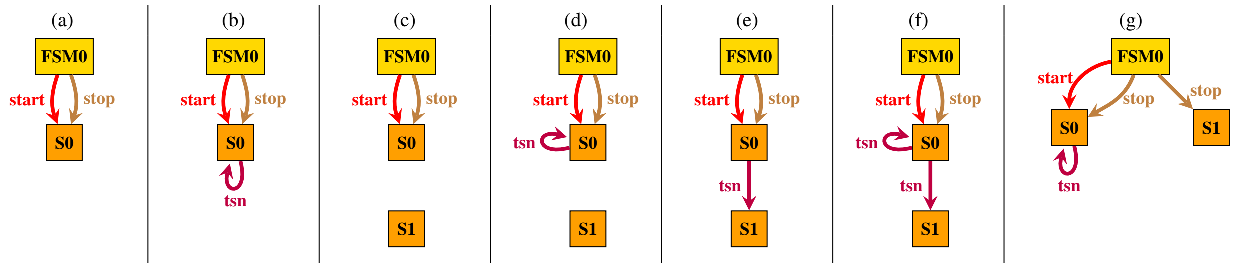


Fig. 6: Revealing Underconstrained Faults Using the Analyzer

users can elect to skip these largely redundant scenarios as desired. As a result, efficiently navigating scenarios which utilize integers is another strength of Hawkeye.

2) Underconstrained Faults: Underconstained faults are revealed when a user enumerates scenarios and notices that the behavior displayed is incorrect. Using Alloy's default enumeration, how many scenarios a user needs to inspect is random, as the faulty behavior can be revealed at any point. Moreover, the user needs to be able to identify that the behavior shown in the scenario is in fact faulty. Using Hawkeye, the user may target behavior that leads to them accidentally revealing a fault. In the process, the user may end up exploring more or less scenarios than Alloy's default enumeration. However, because the user is seeking out specific behavior in the scenario, the user is likely notice quickly if any faulty behavior unexpectedly occurs.

To illustrate, consider the real-world faulty model in Figure 5 (a), which is also from a new Alloy user [36]. The user is trying to specify that for a finite state machine, there is only one start and one stop state. However, the constraints written allow for multiple start and stop states. Specifically, for the start state, the use of "`some`" instead of "`one`" on line 4 allows for there to be more that one start state. The user then adds lines 5-6 to try and say that if any two `State` atoms are equal to the start state, then they are the same `State`. Unfortunately, if there is more the one start state, then the formula "`s1 = FSM.start`" is false. As a result, the user's implication formula is true but not for the user's intended reason, which is what allows for erroneous behavior to be generated. The same holds for stop.

Figure 5 (b) shows the first scenario displayed to the user. Based on this behavior, the user may ask Hawkeye to enumerate a scenario with a different `State` and `tsn` sets to view a larger, more populated finite state machine. This produces the scenario in Figure 5 (c) which displays the faulty behavior. Therefore, using Hawkeye, we can reveal this faulty behavior in one enumeration. The user was expecting to see more state transitions, but is instead presented with multiple stop states. Using Alloy's enumeration, the user can discover faulty behavior after the 6th enumeration, as seen in Figure 6. Due to the order being dependent on the SAT solver, the set of scenarios may change with other configurations and on other machines. Once the user has enumerated the first six scenario in Figure 6 (a) - (f), the user may feel confident enough to stop enumeration. In particular, the scenarios in Figure 6 (a) - (f) show valid behavior for scenarios with multiple states and a variety of state transitions. Therefore, the user may not have any motivation to continue enumeration to Figure 6 (g).

VII. Related Work

Our technique is closely related to techniques which looks to reduce the number of scenarios enumerated by the Analyzer. One traditional approach is symmetry breaking, where constraints are added to the formula to remove isomorphic solutions, which are solutions that share the same shape [3], [29], [14]. Beyond symmetry breaking, several past projects improve scenario enumeration by trying to narrow what scenarios are generated using a specific criteria, e.g., minimality [23], field exhaustiveness [24], and coverage [32], [25]. All of these techniques reduce the number of scenarios by applying some criteria across the entire enumeration. However, the order of scenarios generated is still dictated by the SAT solver's order of discovery. Our approach is orthogonal to these techniques and can work in tandem with their enumeration strategies to take advantage of their reduction in the scenario search space while Hawkeye provides guidance to the order of enumeration.

Another closely related to our work is Seabs, a recent enhancement to Alloy which introduces support for abstract functions [30]. Abstract functions allows users to define data abstractions that specify how scenarios must differ for an executed command. Although abstract functions also enable users to provide information to guide enumeration, abstract functions are a universal goal that impacts the entire collection of scenarios that gets generated and can support broader goals than Hawkeye such as enumerating different transitive closures over an expression [34]. In contrast, our enumeration technique enables users to make on-the-fly decisions that can change from one scenario to the next. Moreover, Hawkeye allows users to express both what elements of a scenario to preserve and discard while abstract functions only allow users to outline how scenarios should differ.

In general, beyond Alloy, researchers have focused on improving scenario enumeration strategies, .e.g. dedicated search [2], mixing of generators and solvers [11], [15], solver-aided languages [26], and sampling [20], [7]. We believe that improvements to the scenarios that get generated by the Analyzer could be refined by some of these approaches, and further combined with Hawkeye. Additionally, researchers have developed novel verification efforts which utilize scenario enumerating toolsets, e.g. automated test input generation [19], and model counting for reliability analysis [9]. In future work, we plan to explore how Hawkeye's enumeration strategy can benefit the broader adoption of these automated verification efforts by allowing the user to explore and have some control over the scenarios used. In particular, TestEra and Korat, which already utilizes elements of Alloy [19], [6].

VIII. Conclusion

Scenario finding toolsets are only valuable if they can generate a feasible number of important scenarios for the user. While the Analyzer produces a robust, exhaustively bounded collection of scenarios, users do not currently have native support within the Analyzer to guide enumeration outside of providing a bound on the universe of discourse. To address this limitation, we introduce Hawkeye, a tool for user-directed enumeration of Alloy scenarios. Hawkeye empowers users to take control of what scenario the Analyzer generates next, allowing users to quickly build a collection of scenarios that is highly valuable to them. Specifically, users can control whether they view a new scenario that is closely related to or vastly different from the current scenario. Our evaluation of Hawkeye shows that the change in enumeration has a minimal overhead. Additionally, we highlight an important use case of Hawkeye: helping users discover if their model is faulty.

Acknowledgments

This work was supported by NSF grant no. CFF-2123341.

References

[1] Alloy analyzer Website. http://alloytools.org. 2019.

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.

[3] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In *AAAI-92 Workshop on Tractable Reasoning*, 1992.

[4] CryptoMiniSat Website. https://www.msoos.org/cryptominisat5/. 2019.

[5] N. Danas, T. Nelson, L. Harrison, S. Krishnamurthi, and D. J. Dougherty. User studies of principled model finder output. In *SEFM*, 2017.

[6] N. Dini, C. Yelen, Z. Alrmaih, A. Kulkarni, and S. Khurshid. Korat-API: A framework to enhance Korat to better support testing and reliability techniques. In *SAC*, 2018.

[7] R. Dutra, J. Bachrach, and K. Sen. SMTSampler: Efficient stimulus generation from complex SMT constraints. In *ICCAD*, 2018.

[8] N. Een and N. Sorensson. An extensible SAT-solver. In *SAT*, 2003.

[9] A. Filieri, C. S. Pasareanu, and W. Visser. Reliability analysis in Symbolic PathFinder. In *ICSE*, 2013.

[10] M. F. Frias, J. P. Galeotti, C. G. L. Pombo, and N. M. Aguirre. DynAlloy: Upgrading Alloy with actions. In *ICSE*, 2005.

[11] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *ICSE*, 2010.

[12] Glucose Website. https://www.labri.fr/perso/lsimon/glucose/. 2019.

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

[14] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson. A case for efficient solution enumeration. In *SAT*, 2003.

[15] I. Kuraj, V. Kuncak, and D. Jackson. Programming with enumerable sets of structures. In *OOPSLA*, 2015.

[16] Lingeling Website. http://fmv.jku.at/lingeling/. 2019.

[17] S. Maoz, J. O. Ringert, and B. Rumpe. CD2Alloy: Class diagrams analysis using Alloy revisited. In *MODELS*, 2011.

[18] S. Maoz, J. O. Ringert, and B. Rumpe. CDDiff: Semantic differencing for class diagrams. In *ECOOP*, 2011.

[19] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, 2001.

[20] K. S. Meel, M. Y. Vardi, S. Chakraborty, D. J. Fremont, S. A. Seshia, D. Fried, A. Ivrii, and S. Malik. Constrained sampling and counting: Universal hashing meets SAT solving. In *Beyond NP, Papers from the 2016 AAAI Workshop*, 2016.

[21] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *LISA*, 2010.

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

[23] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Aluminum: Principled scenario exploration through minimality. In *ICSE*, 2013.

[24] P. Ponzio, N. Aguirre, M. F. Frias, and W. Visser. Field-exhaustive testing. In *FSE*, 2016.

[25] S. Porncharoenwase, T. Nelson, and S. Krishnamurthi. CompoSAT: Specification-guided coverage for model finding. In *FM*, 2018.

[26] T. Ringer, D. Grossman, D. Schwartz-Narbonne, and S. Tasiran. A solver-aided language for test input generation. *OOPLSA*, 2017.

[27] 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.

[28] SAT4J Website. https://www.sat4j.org/. 2019.

[29] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *SAT*, 2001.

[30] A. Sullivan, D. Marinov, and S. Khurshid. Solution enumeration abstraction - A modeling idiom to enhance a lightweight formal method. In *ICFEM*, 2019.

[31] A. Sullivan, K. Wang, S. Khurshid, and D. Marinov. Evaluating state modeling techniques in alloy. In *SQAMIA*, 2017.

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

[33] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.

[34] C. Trippel, D. Lustig, and M. Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests. In *MICRO*, 2018.

[35] K. Wang, A. Sullivan, and S. Khurshid. Automated model repair for Alloy. In *ASE*, 2018.

[36] K. Wang, A. Sullivan, and S. Khurshid. Fault localization for declarative models in Alloy. In *ISSRE*, 2020.