# Evaluating State Modeling Techniques in Alloy

ALLISON SULLIVAN, KAIYUAN WANG, SARFRAZ KHURSHID, University of Texas at Austin, USA
DARKO MARINOV, University of Illinois at Urbana-Champaign, USA

Software models help develop higher quality systems. The declarative language Alloy and its accompanying automatic analyzer embody a method for developing software models. Our focus in this paper is Alloy models of systems where different operations may mutate the system state, e.g., addition of an element to a sorted container. Researchers have previously used two techniques for modeling state and state mutation in Alloy, but these techniques have not been compared to each other. We propose a third technique and evaluate all these three techniques that embody conceptually different modeling approaches. We use four core subjects, which we model using each technique. Our primary goal is to *quantitatively* evaluate the techniques by considering the runtime for solving the ensuing SAT formulas. We also discuss practical tradeoffs among the techniques.

Categories and Subject Descriptors: I.6.4 [**Simulation and Modeling**] Model Validation and Analysis; D.2.5 [**Software Engineering**] Testing and Debugging; D.2.4 [**Software Engineering**] Software/Program Verification

Additional Key Words and Phrases: Alloy, state modeling, SAT solving, empirical evaluation, predicate parameterization

## 1. INTRODUCTION

Building and analyzing software models plays an important role in developing higher quality software systems. The Alloy tool-set – including the declarative language Alloy and its accompanying automatic analysis engine called Alloy analyzer – embodies a method for developing software models [Jackson 2006]. The Alloy language is a relational, first-order logic with transitive closure that allows succinct formulation of complex structural properties. The Alloy analyzer performs *scope-bounded* analysis of Alloy formulas using off-the-shelf Boolean satisfiability (SAT) solvers. The analyzer can generate two forms of valuations for the relations in the model: (1) *instances*, i.e., valuations such that the formulas hold; and (2) *counterexamples*, i.e., valuations such that the negation of the formulas holds. The analyzer enables Alloy users to not only validate their models but also use the tool-set as a basis for various forms of software analyses.

Our focus in this paper is Alloy models of systems where different operations may mutate the system state, e.g., addition of an element to a sorted container. Researchers have used at least two techniques for modeling state and state mutation in Alloy [Jackson and Vaziri 2000; Jackson and Fekete 2001; Marinov and Khurshid 2001; Taghdiri 2003; Frias et al. 2005]. One common technique, which we call *additional state type*, is to introduce a set of state atoms and increase the arity of each relation to add a new state type [Jackson and Fekete 2001; Taghdiri 2003; Frias et al. 2005]. Another common technique, which we call *relation duplication*, is to duplicate relations in the model such that one set of relations represents one state, say pre-state, and another set identifies another state, say post-state [Jackson and Vaziri 2000; Marinov and Khurshid 2001]. A shared intuition at the basis of these techniques is to (explicitly or implicitly) create a representation of each

This research was partially supported by the US NSF Grant Nos. CNS-1239498, CCF-1319688, CCF-1409423, and CCF-1421503.

Authors' addresses: Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, USA; Darko Marinov, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. Email correspondence: khurshid@utexas.edu


In: Z. Budimac (ed.): Proceedings of the 6th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Application (SQAMIA), Belgrade, Serbia, 11.-13.9.2017, published at http://ceur-ws.org

desired state in the model, and write formulas that constrain specific states individually, or sets of states collectively, e.g., to encode post-conditions that relate pre- and post-states. Despite the common basis, these techniques are technically quite different – not only in terms of syntactic and semantic representation but also in terms of the state spaces that ensue for SAT exploration.

While state modeling techniques have allowed effective applications of Alloy in various domains – including software design [Jackson and Fekete 2001; Taghdiri 2003; Frias et al. 2005], analysis [Jackson and Vaziri 2000; Dennis et al. 2006; Milicevic et al. 2011; Galeotti et al. 2013], testing [Marinov and Khurshid 2001], and security [Kang et al. 2016] – these techniques have not been compared to each other. We propose a third technique, called *predicate parameterization*, and compare all three techniques that embody conceptually different modeling approaches. We use four core subjects that we chose because they represent two broad classes of problems – two subjects are data structures representative of many evaluations done with Alloy [Jackson and Vaziri 2000; Marinov and Khurshid 2001; Galeotti et al. 2013] and two subjects are from the standard Alloy distribution. We are not aware of any common benchmark set of Alloy models for evaluating performance of the Alloy analyzer. We model each subject using each technique. We do not use more or bigger subjects because translating each model from one technique to another currently requires a substantial manual effort. Our primary goal is to *quantitatively* evaluate the techniques by considering the runtime for solving the ensuing SAT formulas. (In other words, we do not consider the asymptotic algorithm complexity but the actual practical performance.) We also discuss practical trade-offs among the techniques.

## 2. TECHNIQUES

This section describes the three state modeling techniques that we evaluate. We first introduce an illustrative example and some basic concepts of Alloy (Section 2.1). We then describe the three techniques and illustrate them using our example (Section 2.2).

### 2.1 Illustrative example and Alloy basics

Consider modeling an acyclic, sorted, singly-linked list with unique elements in Alloy. The following snippet declares the basic Alloy data-types:

```
sig List {
  header: lone Node
}

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

The `sig` declaration introduces a set of *atoms* and optionally declare fields, i.e., relations. The field `header` is a binary relation of type `List` × `Node` and represents the list's first node; `elem` has type `Node` × `Int` and represents the node's integer (`Int`) element; and `link` has type `Node` × `Node` and represents the node's next node. The keyword `lone` declares the binary relation to be a partial function, e.g., each list has at most one header node, and each node has at most one next mode. By default, each binary relation that is declared is a total function, e.g., each node contains exactly one integer element.

Consider expressing acyclicity. The following snippet is an Alloy predicate (`pred`), i.e., a named, parameterized formula that may be invoked elsewhere, which defines acyclicity using universal quantification (`all`):

```
pred Acyclic(l: List) {
  all n: l.header.*link | n !in n.^link
}
```

The operator '`.`' is relational composition; '`*`' is reflexive transitive closure; and '`^`' is transitive closure. Note that '`*`' and '`^`' are used as prefix not suffix operators. The (infix) operator '`!in`' denotes that the left-hand expression is not a *subset* of the right-hand expression. Note that this operator does not denote just "not an *element*" because all Alloy expressions are semantically relations (even if of arity only one, i.e., sets) and not scalar atoms [Jackson 2006]. The expression `l.header.*link` represents the set of all nodes reachable from `l`'s header along `link` (including the header itself). The predicate encodes that for any node `n` in the list, the set of nodes reachable from `n` does not contain `n`, hence there is no cycle.

The following predicate defines sortedness (with unique elements):

```
pred SortedUnique(l: List) {
  all n: l.header.*link | some n.link => n.elem < n.link.elem
}
```

The operator '`=>`' is logical implication. The formula `some n.link` encodes that the expression `n.link` is a non-empty set. The predicate encodes that for any node in the list, if the node has a next node, the elements from the two nodes are in the ascending order; the operator '`<`' is integer comparison.

The following Alloy snippet defines the predicate `RepOk` that is a conjunction of `Acyclic` and `SortedUnique`, and uses the `run` command to instruct the analyzer to create an instance in the scope of 1 list, 3 nodes, and bit-width of 2 for integers:

```
pred RepOk(l: List) {
  Acyclic[l]
  SortedUnique[l]
}

run RepOk for 1 List, 3 Node, 2 int
```

## 2.2 Additional state type

The most widely used technique for modeling state in Alloy is to introduce a new sig, commonly called `State`, and add it to each relation, increasing the relation's arity by one [Jackson and Fekete 2001; Taghdiri 2003; Frias et al. 2005]. For example, the following snippet shows this technique applied to the list declaration:

```
abstract sig State {}

sig List {
  header: Node -> State
}
fact { all l: List | all s: State | lone l.(header.s) }
```

`State` is an abstract sig, i.e., it contains only atoms that are strictly necessary for the constraint solved. The symbol '`->`' denotes the Cartesian product in expressions and adds arity in declarations. The field `header` is now a ternary relation of type `List` × `Node` × `State`, which allows a list to have different nodes as its header in different states. Note that the state need not be the last type; it can be in any position, e.g., in the first position where the `sig State` would have other relations (such as `header` and `link`) as its fields. We use state in the last position because it allows us to preserve the declaration structure of the original model. A `fact` in Alloy is a formula that must always hold. We use a fact to require each list to have at most one header node in each state to conform to the partial function relation in the original model (without state).

The following snippet shows how the predicate `Acyclic` can be written in the presence of state:

```
pred Acyclic(l: List, s: State) {
  all n: l.(header.s).*(link.s) | n !in n.^(link.s)
}
```

Note the new state parameter `s` for which the predicate holds, and also the new composition of each relation with a state to represent the field values in the desired state, e.g., `l.(header.s)` is the header of the list `l` in the state `s`.

Consider next modeling state mutation. This snippet defines removal of the first node from the list:

```
pred RemoveFirst(l: List, s: State, s': State) {
  s != s' -- states are unique
  RepOk[l, s] -- l satisfies RepOk in s
  l.(header.s).*(link.s).(elem.s) - l.(header.s).(elem.s) = l.(header.s').*(link.s').(elem.s')
  RepOk[l, s'] -- l satisfies RepOk in s'
}

run RemoveFirst for 2 State, 1 List, 3 Node, 2 int
```

The predicate has two state parameters: `s` represents the pre-state, and `s'` represents the post-state. The operator '-' is set difference. (The symbol '--' is used for comments.) The predicate encodes that the two states are distinct; `l` satisfies `RepOk` in the pre-state; the set of elements in the pre-state minus the header element in the pre-state is the set of elements in the post-state; and `l` satisfies `RepOk` in the post-state. Figure 1 graphically illustrates an instance for `RemoveFirst`.

Consider next using the analyzer to check whether `RemoveFirst` has a specific property. The following snippet uses an Alloy assertion (`assert`) to encode that `RemoveFirst` implies that the header element in the post-state is the second element from the pre-state:

```
assert PartialCorrectnessOnce {
  all disj s, s': State | all l: List |
    RemoveFirst[l, s, s'] => l.(header.s').(elem.s') = l.(header.s).(link.s).(elem.s)
}

check PartialCorrectnessOnce for 3
```

The keyword `disj` requires `s` and `s'` to be distinct. The command `check` instructs the analyzer to find a counterexample to the named assertion, i.e., `PartialCorrectnessOnce`. However, the analyzer does not find a counterexample for this command in this example for the given scope of 3. (There could exist a counterexample in a larger scope.)

## 2.3 Relation duplication

Another technique for modeling state is to introduce a new copy of declared relations for each state and to model mutation by defining constraints across the relations for different states [Jackson and Vaziri 2000; Marinov and Khurshid 2001]. To illustrate, consider modeling pre-state and post-state for `RemoveFirst`. (In general, there could be more than two states, and the relations would need to be copied multiple times.) The following snippet shows this technique applied to the list declaration:

```
sig List {
  header: lone Node, -- pre-state
  header': lone Node -- post-state
}
```

The mutation of the original header field is now modeled by two relations: `header` that represents the value in the pre-state, and `header'` that represents the value in the post-state.

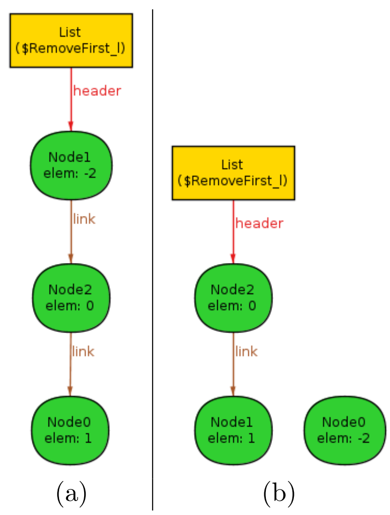


Fig. 1. Example RemoveFirst (a) pre- and (b) post-state visualized using Alloy analyzer

Constraints on the relations are now written over appropriate groups of relations. The following snippet shows how two predicates can be written to represent acyclicity for the two states:

```
pred Acyclic(l: List) { -- for pre-state
  all n: l.header.*link | n !in n.^link
}

pred Acyclic'(l: List) { -- for post-state
  all n: l.header'.*link' | n !in n.^link'
}
```

`Acyclic` represents acyclicity in the pre-state, and `Acyclic'` represents acyclicity in the post-state. Note that each predicate uses relations only from its corresponding state. Similar changes are made for `RepOk` and `RepOk'` (and `SortedUnique` and `SortedUnique'`).

Consider next modeling state mutation. The following snippet defines `RemoveFirst` using this technique:

```
pred RemoveFirst(l: List) {
  RepOk[l] -- RepOk in pre-state
  l.header.*link.elem - l.header.elem = l.header'.*link'.elem'
  RepOk'[l] -- RepOk in post-state
}

run RemoveFirst for 1 List, 3 Node, 2 int
```

Moreover, the following snippet defines the assertion `PartialCorrectnessOnce` using this technique:

```
assert PartialCorrectnessOnce {
  all l: List | RemoveFirst[l] => l.header'.elem' = l.header.link.elem
}

check PartialCorrectnessOnce for 3
```

## 2.4 Parameterization

The third technique we evaluate removes all relation declarations from sig declarations, adds the relations as parameters to all predicates, and adds a new predicate to express all the facts in the model. For example, the declaration of the list signature becomes just the following:

```
sig List {}
```

The following snippet illustrates adding the relations as parameters to the acyclicity predicate:

```
pred Acyclic(l: List, header: List -> Node, elem: Node -> Int, link: Node -> Node) {
  all n: l.header.*link | n !in n.^link
}
```

Similar changes are made for `RepOk` (and `SortedUnique`).

In addition to changing all the existing predicates, a new predicate is added to encode all the facts from the original model. The following snippet illustrates the new predicate, which should be appropriately invoked when commands are executed:

```
pred SigDeclFacts(header: List -> Node, elem: Node -> Int, link: Node -> Node) {
  all l: List | lone l.header
  all n: Node | one n.elem and lone n.link
}
```

In Alloy, '`one`' holds if its expression denotes a singleton set/relation, while '`and`' is the usual conjunction, expressed explicitly. (There is also an implicit conjunction among the formulas on different lines.)

Consider next modeling state mutation. The following snippet defines `RemoveFirst` using this technique:

```
pred RemoveFirst(l: List, header: List -> Node, elem: Node -> Int, link: Node -> Node,
                          header': List -> Node, elem': Node -> Int, link': Node -> Node) {
  RepOk[l, header, elem, link]
  l.header.*link.elem - l.header.elem = l.header'.*link'.elem'
  RepOk[l, header', elem', link']
}

pred RunRemoveFirst(l: List, header: List -> Node, elem: Node -> Int, link: Node -> Node,
                             header': List -> Node, elem': Node -> Int, link': Node -> Node) {
  SigDeclFacts[header, elem, link] and SigDeclFacts[header', elem', link']
  RemoveFirst[l, header, elem, link, header', elem', link']
}

run RunRemoveFirst for 1 List, 3 Node, 2 int
```

In addition to the list parameter, `RemoveFirst` has 6 relations as parameters: 3 for pre-state (`header`, `elem`, and `link`) and 3 for post-state (`header'`, `elem'`, and `link'`). To run `RemoveFirst`, a new predicate `RunRemoveFirst` is introduced and run, which appropriately enforces the facts from the original model (without state). This new predicate `RunRemoveFirst` is not expected to be invoked elsewhere (in another predicate); its only purpose is to enable a run command that conforms to the semantics of facts in Alloy.

The following snippet defines the assertion `PartialCorrectnessOnce` using this technique:

```
assert PartialCorrectnessOnce {
  all l: List | all header: List -> Node | all elem: Node -> Int | all link: Node -> Node |
              all header': List -> Node | all elem': Node -> Int | all link': Node -> Node {
  {
    SigDeclFacts[header, elem, link] and SigDeclFacts[header', elem', link']
  } =>
    RemoveFirst[l, header, elem, link, header', elem', link'] => l.header'.elem' = l.header.link.elem
}}

check PartialCorrectnessOnce for 3
```

The structure of the assertion assumes the facts, once again to conform to the semantics of facts in Alloy.

## 3. EVALUATION

We use four core subjects – two data structures and two subjects from the standard Alloy distribution – as base models, providing us 11 constraint-solving problems with different complexities to quantitatively compare the three techniques:

(1) **Singly-linked list**, our running example; we derive four problems: (a) create an instance for removing the first element (`RemoveFirst`), our running example; (b) create an instance for removing the first element twice (`TwiceRemoveFirst`), which requires three states (unlike our running example that used only two states); (c) check that `RemoveFirst` implies that the header element in post-state is the second element in the pre-state (`PartialCorrectnessOnce`); and (d) check that if a list has two or more elements, removing the first element twice implies the number of nodes in the list reduces by two (`PartialCorrectnessTwice`);

(2) **Binary search tree**, we derive four problems: (a) create an instance for adding a given element to the tree (`Add`); (b) create an instance for removing a given element from the tree (`Remove`); (c) check that adding an element that is not in the tree followed by removing the same element leaves the set of elements originally in the tree unchanged (`AddRemoveNoOp`); and (d) check that two is the difference in the number of nodes between (i) adding a new element to the tree versus (ii) removing an existing element from the tree (`AddRemoveComparison`).

Table I. State techniques comparison; P.V. and Cl. are primary variables and clauses, resp., in the SAT formula

| Model | Problem/Command | Additional state type | | | Relation duplication | | | Parameterization | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **T[ms]** | **P.V.** | **Cl.** | **T[ms]** | **P.V.** | **Cl.** | **T[ms]** | **P.V.** | **Cl.** |
| List | RemoveFirst | 102 | 59 | 2706 | 47 | 77 | 2104 | 18 | 53 | 1956 |
| | TwiceRemoveFirst | 83 | 89 | 4514 | 15 | 77 | 2846 | 24 | 77 | 2858 |
| | PartialCorrectnessOnce | 13949 | 216 | 13709 | 11267 | 207 | 10298 | 2223 | 141 | 10001 |
| | PartialCorrectnessTwice | 44555 | 219 | 18391 | 30535 | 207 | 14032 | 5563 | 207 | 14201 |
| Tree | Add | 27 | 81 | 3162 | 36 | 108 | 2441 | 23 | 75 | 1983 |
| | Remove | 8 | 81 | 3162 | 42 | 108 | 2441 | 15 | 75 | 1983 |
| | AddRemoveNoOp | 31 | 262 | 13583 | 53 | 250 | 9313 | 22 | 250 | 9312 |
| | AddRemoveComparison | 230366 | 278 | 13698 | 195216 | 266 | 9428 | 123419 | 266 | 9427 |
| Farmer | solvePuzzle | 29 | 66 | 2239 | 30 | 58 | 1850 | 27 | 66 | 2246 |
| | NoQuantumObjects | 45 | 78 | 2441 | 23 | 62 | 2121 | 51 | 66 | 2321 |
| Dijkstra | Deadlock | 134 | 260 | 826 | 61 | 255 | 592 | 49 | 255 | 605 |
| | ShowDijkstra | 87 | 47 | 1520 | 26 | 42 | 1491 | 13 | 42 | 1505 |
| | DijkstraPreventsDeadlocks | 546 | 210 | 9498 | 392 | 205 | 9321 | 225 | 205 | 9322 |

(3) **Farmer**, the classic puzzle on crossing the river, which describes that a farmer wants to move a fox, a chicken, and a bag of grain from one bank of a river to the other bank without losing any of them. This model comes with the standard Alloy distribution, where it already has a `State` signature that represents the object status for both river banks every time the farmer moves. The state is the first type in the corresponding relations. The model includes two problems: (a) solve the puzzle (`solvePuzzle`); and (b) check that no object is at more than one place at the same time (`NoQuantumObjects`).

(4) **Dijkstra**, a model of Dijkstra's mutual exclusion for processes, which is also in the standard Alloy distribution; similar to Farmer, state is the first type in the relations used to model mutation. The model includes three problems: (a) create an instance that shows a deadlock (`Deadlock`); (b) try to find a deadlock instance where the process mutexes are grabbed and released based on the Dijkstra algorithm (`ShowDijkstra`); and (c) directly check that the Dijkstra algorithm prevents deadlocks (`DijkstraPreventsDeadlocks`).

Table I shows the experimental results. For each model, we list the executed commands. The scope for **List** (resp., **Tree**) has values as shown in the example: 1 list (resp., 1 tree), 3 nodes, and bit-width of 2 for integers. The scope for **Farmer** has 8 states and 4 fixed objects (Farmer + Fox + Chicken + Grain). The scope for **Dijkstra** has `5 State, 5 Process, 5 Mutex` for `Deadlock`; `5 State, 2 Process, 2 Mutex` for `ShowDijkstra`, and `5 State, 5 Process, 4 Mutex` for `DijkstraPreventsDeadlocks`. We leave it as future work to experiment with different scopes and models.

For each modeling technique, we tabulate time (in milliseconds) to solve the resulting SAT formula ($T[ms]$), the number of primary variables ($P.V.$), and the number of clauses ($Cl.$) in the SAT formula. All the experiments were run on an Intel Celeron CPU N3060 1.60GHz x 2 processor with 1.8GB of memory using Alloy 4 (http://alloy.mit.edu/alloy/downloads/alloy4.jar). We initially tried to use Alloy 4.2, the latest Alloy release, but encountered an anomalous behavior: our list and tree models using parameterization created SAT formulas with 0 primary variables and 0 clauses; we confirmed that this is a bug in Alloy 4.2.

For the four problems where the solving time exceeds 500ms for any of the techniques, *parameterization provides the most efficient solving*, followed by relation duplication, and then additional state type. The performance difference is the greatest for `PartialCorrectnessTwice` in list, where parameterization provides a speedup of 8X over additional state type. While the time for SAT solving is determined by the complexity, not just the size, of the SAT formula, one reason for the performance difference can be the size. For each of these four problems, the number of *primary variables* is the smallest for parameterization, which is the same as the number for duplication with one exception (`PartialCorrectnessOnce`). Moreover, the number of *clauses* for parameterization and duplication is quite close for these four problems but noticeably smaller than the number for additional state type. Overall, parameterization enables a tight encoding that leads to efficient analysis for these problems.

For the problems where the solving time is below 500ms for all techniques, the difference in time among the techniques is not practically relevant, so any can be used for just one small problem. However, the techniques do differ, and more precise and extensive measurements would be needed to find the best technique for analyzing a large number of small problems. In particular, it would be important to understand the cases where parameterization is not the best technique.

Quantitatively, the models created using parameterization are the fastest to solve. Qualitatively, however, the models created using additional state type are most readable due to two reasons: (1) state can be conveniently referred to, e.g., a quantified formula can directly be written over the set of states; and (2) the type declaration structure of the original model (without state) can be largely preserved. The models with relation duplication are burdensome for (manual) maintenance because the predicates have to exist in multiple copies (e.g., `Acyclic` and `Acyclic'`). The models with parameterization require unwieldy predicate signatures because all predicates are parameterized; in addition, facts need to be explicitly handled in a special way. We note that different techniques are best suited to different purposes, e.g., additional state type for manual modeling, and relation duplication and parameterization for automated analyses where the models are mechanically generated. Indeed, future work should consider automatic translations that map a model built using one technique to conform to another technique for more efficient back-end analysis.

## 4. CONCLUSIONS

The Alloy software modeling tool-set has been effectively used in software design, analysis, and testing. Our focus in this paper was on comparing Alloy modeling techniques for systems where different operations may mutate the system state. Over the years, researchers have use at least two techniques for modeling state and state mutation in Alloy, but these techniques were not previously compared to each other. We proposed a third technique and evaluated all three techniques that embody conceptually different modeling approaches. We used four core subjects, which we model using each technique. The results show that the models created using the *parameterization* technique are the fastest to solve. However, such models are hard to write manually and should be automatically derived from different models.

## REFERENCES

Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular Verification of Code with SAT. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 109–120.

Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. 2005. DynAlloy: Upgrading Alloy with Actions. In *Proc. 27th International Conference on Software Engineering (ICSE)*. 442–451.

Juan P. Galeotti, Nicolás Rosner, Carlos G. López Pombo, and Marcelo F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *IEEE Trans. Software Eng.* 39, 9 (2013), 1283–1307.

Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press.

Daniel Jackson and Alan Fekete. 2001. Lightweight Analysis of Object Interactions. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*. 492–513.

Daniel Jackson and Mandana Vaziri. 2000. Finding Bugs with a Constraint Solver. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 14–25.

Eunsuk Kang, Aleksandar Milicevic, and Daniel Jackson. 2016. Multi-representational Security Analysis. In *Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 181–192.

Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*. 22–31.

Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. 2011. Unifying Execution of Imperative and Declarative Code. In *Proc. 33rd International Conference on Software Engineering (ICSE)*. 511–520.

Mana Taghdiri. 2003. *Lightweight Modelling and Automatic Analysis of Multicast Key Management Schemes.* Master's thesis. Massachusetts Institute of Technology.