Symbolic Execution of Alloy Models

Junaid Haroon Siddiqui and Sarfraz Khurshid

The University of Texas at Austin

Abstract. Symbolic execution is a technique for systematic exploration of program behaviors using symbolic inputs, which characterize classes of concrete inputs. Symbolic execution is traditionally performed on imperative programs, such as those in C/C++ or Java. This paper presents a novel approach to symbolic execution for declarative programs, specifically those written in Alloy – a first-order, declarative language based on relations. Unlike imperative programs that describe *how* to perform computation to conform to desired behavioral properties, declarative programs describe *what* the desired properties are, without enforcing a specific method for computation. Thus, symbolic execution does not directly apply to declarative programs the way it applies to imperative programs. Our insight is that we can leverage the fully automatic, SAT-based analysis of the Alloy Analyzer to enable symbolic execution of Alloy models – the analyzer generates instances, i.e., valuations for the relations in the model, that satisfy the given properties and thus provides an execution engine for declarative programs. We define symbolic types and operations, which allow the existing Alloy tool-set to perform symbolic execution for the supported types and operations. We demonstrate the efficacy of our approach using a suite of models that represent structurally complex properties. Our approach opens promising avenues for new forms of more efficient and effective analyses of Alloy models.

1 Introduction

Symbolic execution [5, 14] is a technique first presented over three decades ago for systematic exploration of behaviors of imperative programs using symbolic inputs, which characterize classes of concrete inputs. The key idea behind symbolic execution is to explore (feasible) execution paths by building *path conditions* that define properties required of inputs to execute the corresponding paths. The rich structure of path conditions enables a variety of powerful static and dynamic analyses. However, traditional applications of symbolic execution have largely been limited to small illustrative examples, since utilizing path conditions in automated analysis requires much computation power, particularly for non-trivial programs that have long execution paths with complex control flow. During recent years, many advances has been made in constraint solving technology [7] and additionally, raw computation power has increased substantially. These advancements have led to a resurgence of symbolic execution, and new variants that perform *partial* symbolic execution have become particularly popular for systematic bug finding [4] in programs written in commonly used languages such as C/C++, C#, and Java.

While symbolic execution today lies at the heart of some highly effective and efficient approaches for checking imperative programs, the use of symbolic execution in declarative programs is uncommon. Unlike imperative programs that describe *how* to perform computation to conform to desired behavioral properties, declarative programs describe *what* the desired properties are, without enforcing a specific method for computation. Thus, symbolic execution, or execution per se, does not directly apply to declarative programs the way it applies to imperative programs.

This paper presents a novel approach to symbolic execution for declarative programs written in the Alloy modeling language [11]. Alloy is a first-order declarative logic based on sets and relations, and is supported by its fully automatic, SAT-based analyzer. The Alloy tool-set is rapidly gaining popularity for academic research and teaching as well as for designing dependable software in industry. The powerful analysis performed by the analyzer make Alloy particularly attractive for modeling and checking a variety of systems, including those with complex structural constraints – SAT provides a particularly efficient analysis engine for such constraints.

Our insight into symbolic execution for Alloy is that path conditions in symbolic execution, which by definition are constraints (on inputs), can play a fundamental role in effective and efficient analysis of declarative programs, which themselves are constraints (that describe "what"). The automatic analysis performed by the Alloy tool-set enables our insight to form the basis of our approach. Given an Alloy model, the analyzer generates *instances*, i.e., concrete valuations for the sets and relations in the model, which satisfy the given properties. Thus, the analyzer, in principle, already provides an *execution engine* for declarative programs, which bears resemblance to concrete execution of imperative programs. Indeed, a common use of the analyzer is to *simulate* Alloy predicates and iterate over concrete instances that satisfy the predicate constraints [11]. The novelty of our work is to introduce *symbolic execution* of Alloy models, which is inspired by traditional symbolic execution for imperative programs. Specifically, we introduce symbolic types and symbolic operators for Alloy, so that the existing Alloy Analyzer is able to perform symbolic execution for the supported types and operations. To illustrate, symbolically simulating an Alloy predicate using our approach allows generating a *symbolic instance* that consists of a concrete valuation, similar to a traditional Alloy instance, as well as a symbolic valuation that includes a constraint on symbolic values, similar to a path condition.

We demonstrate the efficacy of our approach using a suite of models that represent a diverse set of constraints, including structurally complex properties. Our approach opens promising avenues for new forms of more efficient and effective analyses of Alloy models. For example, our approach allows SAT to be used to its optimal capability for structural constraint solving, while allowing solving of other kinds of constraints to be delegated to other solvers. As another example, our approach allows Alloy users to view multiple instances simultaneously without the need for enumeration through repeated calls to the underlying solver: a symbolic instance represents a class of concrete instances.

This paper makes the following contributions:

- **Symbolic execution for declarative programs.** We introduce the idea of symbolic execution for declarative programs written in analyzable notations, similar to symbolic execution of imperative programs.
- **Symbolic execution for Alloy models.** We present our approach for symbolic execution of Alloy, and provide an extensible technique to support various symbolic types and operators.
- **Demonstration.** We use a suite of small but complex declarative models to demonstrate the efficacy of our approach and the promise it holds in laying the foundation of novel methodologies for automated analysis of declarative programs.

2 Background & Illustrative Example

This sections presents background on symbolic execution and Alloy and an example of symbolic execution for Alloy technique using a sorted linked list.

2.1 Symbolic Execution Basics

Forward symbolic execution is a technique for executing a program on symbolic values [14]. There are two fundamental aspects of symbolic execution: (1) defining semantics of operations that are originally defined for concrete values and (2) maintaining a *path condition* for the current program path being executed – a path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path.

As an example, consider the following program that returns the absolute value of its input:

```
       static int abs(int x) {
L1.      int result;
L2.      if (x < 0)
L3.        result = 0 - x;
L4.      else result = x;
L5.      return result;      }
```

To symbolically execute this program, we consider its behavior on a primitive integer input, say `X`. We make no assumptions about the value of `X` (except what can be deduced from the type declaration). So, when we encounter a conditional statement, we consider both possible outcomes of the condition. To perform operations on symbols, we treat them simply as variables, e.g., the statement on `L3` updates the value of `result` to be `0-X`. Of course, a tool for symbolic execution needs to modify the type of `result` to note updates involving symbols and to provide support for manipulating expressions, such as `0-X`.

Symbolic execution of the above program explores the following two paths:

```
path 1:    [X < 0] L1 -> L2 -> L3 -> L5
path 2:    [X >= 0] L1 -> L2 -> L4 -> L5
```

Note that for each path that is explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these two paths, symbolic execution explores both.

2.2 Alloy Basics

Alloy is a first-order relational language [11]. An Alloy specification is a sequence of paragraphs that either introduce new types or record constraints on *fields* of existing types. Alloy assumes a universe of atoms partitioned into subsets, each of which is associated with a basic type. Details of the Alloy notation and of the Alloy Analyzer can be found in [11].

Acyclic lists can be modeled in Alloy with the following specification (called `SortedList` for consistency with the example in the following section):

```
one sig SortedList {
  header: lone Node,
  size: Int }
sig Node {
  data: Int,
  nextNode: lone Node }
pred Acyclic(l: SortedList) {
  all n: l.header.*nextNode | n !in n.^nextNode }
```

The *signature* declarations `SortedList` and `Node` introduce two uninterpreted types, along with functions `header : SortedList` $\rightarrow$ `Node`, `size : SortedList` $\rightarrow$ `Int`, `data : Node` $\rightarrow$ `Int`, and `nextNode : Node` $\rightarrow$ `Node`. `header` and `nextNode` are partial functions, indicated by the declaration `lone`.

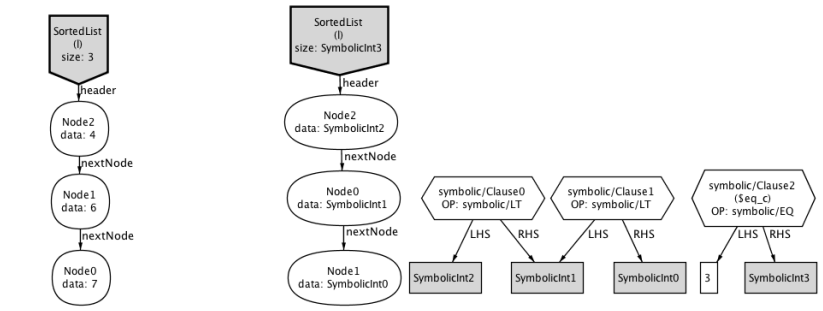
The Alloy *predicate* Acyclic, when invoked, constrains its input `l` to be acyclic. The dot operator '`.`' represents relational image, '`~`' represents transpose, '`^`' represents transitive closure, and '`*`' denotes reflexive transitive closure.

The quantifier `all` stands for universal quantification. For instance, the constraint `all n: l.header.*nextNode | F` holds if and only if evaluation of the *formula* `F` holds for each atom in the transitive closure of `nextNode` starting from `l.header`. Formulas within curly braces are implicitly conjoined. The quantifier `lone` stands for "at most one". There are also quantifiers `some` and `no` with the obvious meaning.

Given an Alloy specification, the Alloy Analyzer automatically finds *instances* that satisfy the specification, i.e., the valuations of relations and signatures that make all the facts in the specification true. Alloy Analyzer finds instances within a pre-specified *scope* – the maximum number of atoms in each basic signature. Alloy Analyzer can also enumerate all non-isomorphic instances.

2.3 Illustrative Example: Symbolic Execution for Alloy

This section presents an example of symbolic execution of Alloy formulas using a sorted linked list. In Section 2.2 we presented the Alloy specification for a linked list. To make it into a sorted linked list, we use the following predicate.



(a) Concrete (b) Symbolic

Fig. 1. Visualizing a sorted linked list with three nodes

```
pred RepOk(l: SortedList) {
  all n: l.header.*nextNode| n !in n.^nextNode -- acyclicity
  #l.header.*nextNode = l.size -- size ok
  all n: l.header.*nextNode |
    some n.nextNode => n.data < n.nextNode.data } -- sorted
```

The Alloy Analyzer can be used on this model to find instances of a sorted linked list. We add the following commands to our model to test the `RepOk` predicate for three nodes.

```
fact { SortedList.header.*nextNode = Node } -- no unreachable Node
run RepOk for 3 -- maximum 3 atoms of each kind
```

As a result of executing this model, Alloy Analyzer produces an instance. The user can get more and more instances by clicking next. One example instance is shown in Fig. 1(a). This sorted linked list represents the sequence $\langle 4, 6, 7 \rangle$. The Alloy Analyzer produces many more instances with three nodes with different sorted arrangements of integers in the domain of Alloy integers.

Symbolic execution of Alloy (Section 3) is a technique to produce instances with symbolic variables and a set of constraints over those symbolic variables. These individual constraints are called *clauses* in our models. The technique is implemented as (1) Alloy library module, (2) a set of guidelines for the user on how to write their Alloy formulas, (3) a set of mechanically generated rules, and finally (4) a mechanism to invoke the Alloy Analyzer. To use symbolic execution of Alloy, the user writing the model has to include the `symbolic` module:

```
open symbolic
```

The user changes any uses of `Int` they wants to make symbolic to `SymbolicInt`. The updated signature declarations for our example look like this:

```
one sig SortedList {
  header: lone Node,
  size: SymbolicInt }
```

```
sig Node {
  data: SymbolicInt,
  nextNode: lone Node }
```

Lastly, the user changes any operations performed on the symbolic variables to use the predicates provided by the `symbolic` module (e.g. `eq`, `lt`, `gt`, etc.). We follow the predicate names introduced by the Alloy 4.2 release candidate in its `integer` module for concrete operations on integers. If the user uses these predicates, no changes are required for predicate invocation. The Alloy Analyzer's type checking finds out whether a symbolic operation is needed or a concrete operation. The updated `RepOk` predicate looks like this:

```
pred RepOk(l: SortedList) {
  all n: l.header.*nextNode| n !in n.^nextNode -- acyclicity
  (#l.header.*nextNode).eq[l.size] -- size ok
  all n: l.header.*nextNode |
    some n.nextNode => (n.data).lt[n.nextNode.data] } -- sorted
```

As the next step, the Alloy module is transformed and a new `fact` is mechanically generated. This fact ensures that the symbolic integers used are all unique. For sorted linked list, this fact is:

```
fact {
  #SymbolicInt = (#SortedList).plus[#Node]
  SymbolicInt = SortedList.size + Node.data }
```

Finally, when this updated model is run through Alloy Analyzer, models with a set of constraints on these symbolic integers are generated. An example instance with three nodes is given in Fig. 1(b). This time, however, it is the only instance with three nodes. Other instances either have fewer or more nodes. This helps the user visualize the model in a more efficient manner. Also a symbolic instance more explicitly states the relationship between data nodes.

3 Symbolic execution of Alloy formulas

This section presents the four key parts of our approach: (1) Alloy library module that introduces symbolic variables and operations on them as well as a representation for clauses that define constraints on symbolic fields, (2) changes required in the user model to introduce symbolic fields, (3) mechanically generated facts that enable consistent usage of symbolic values, and (4) Alloy Analyzer usage to restrict any redundant clauses from being generated.

3.1 Symbolic Alloy module

This section presents the Alloy module that enables symbolic execution. The module starts by the module declaration and a few signatures:

```
module symbolic
```

```
abstract sig Expr {}
sig SymbolicInt, SymbolicBool extends Expr {}

abstract sig RelOp {}
one sig lt, gt, lte, gte, eq, neq, plus, minus extends RelOp {}
```

`Expr` atoms represent expressions that can be symbolic variables or expressions on symbolic variables and plain integers. `RelOp` are single atoms (because of the `one` modifier) that represents a few binary operations we demonstrate. Next we define the `Clause` atom, which is an expression combining two symbolic variables, standard Alloy integers, or expressions.

```
abstract sig Clause extends Expr {
  LHS: Expr+Int,
  OP: RelOp,
  RHS: Expr+Int }
```

Next, we have a set of predicates that require certain clauses to exist. For example the following `lt` and `eq` predicates would require that appropriate `Clause` atoms must exist. These `Clause` atoms in the final output show us the relationship enforced on symbolic variables in the model.

```
pred lt(e1: Expr+Int, e2: Expr+Int) {
  some c: Clause | c.LHS = e1 && c.OP = LT && c.RHS = e2 }
pred eq(e1: Expr+Int, e2: Expr+Int) {
  some c: Clause | c.LHS = e1 && c.OP = EQ && c.RHS = e2 }
```

Similar predicates exist for all supported operations and Alloy functions exist to combine `plus` and `minus` operators to form more complex expressions.

3.2 User modifications to Alloy model

This section describes the changes required of the user in their model. Some such changes were discussed in Section 2.3 in the context of a sorted linked list.

The first change is a call to use the `symbolic` module. This imports the library signatures, predicates, and functions discussed in the previous section.

```
open symbolic
```

Next the user changes `Int` to `SymbolicInt` and `Bool` to `SymbolicBool`. These are the only primitive types supported by the Alloy Analyzer and we enable symbolic analysis for both of them.

Lastly, the user has to change all operations on symbolic variables to use one of the predicates or functions in the `symbolic` module. However, the names we used are the same as those used in the built-in Alloy `integer` module. The new recommended syntax of Alloy 4.2 release candidate is already to use such predicates. Specifically, for `plus` and `minus` predicates, the old syntax is no longer allowed. The `+` and `-` operators exclusively mean set union and set difference now.

We follow the lead of this predicate-based approach advocated in the Alloy 4.2 release candidate and support `eq`, `neq`, `lt`, `gt`, `lte`, `gte`, `plus`, and `minus` in our

`symbolic` module. If the user is using old Alloy syntax, he has to change to the new syntax as follows:

```
a = b  ⇒  a.eq[b]
a < b  ⇒  a.lt[b]
a > b  ⇒  a.gt[b]
a + b  ⇒  a.plus[b]
a - b  ⇒  a.minus[b]
```

The `plus` and `minus` operations in our `symbolic` library come in two forms: as a predicate and as a function. The predicate requires the clause to exist and the function returns the existing clause. For example, to convert an expression `a+b>c` the user first converts it to new syntax i.e. `(a.plus[b]).gt[c]`. Then he adds the plus operation as a separate predicate as well i.e. `a.plus[b] && (a.plus[b]).gt[c]`. The compiler recognizes the first invocation as a predicate that requires a new clause to exist and the second invocation as returning that clause. If the predicate is omitted, the function returns no clause and no satisfying model is found. We include two case studies that show how it is used (Section 4.3 and Section 4.4).

3.3 Mechanically generated facts

This section presents the Alloy facts that our technique mechanically generates to ensure soundness of symbolic execution. These facts ensure that symbolic variables are not shared among different objects. For example, two `Node` atoms cannot point to the same `SymbolicInt` atom as `data`. Otherwise, we cannot distinguish which nodes's symbolic variable a `Clause` is referring to. Note that this does not prevent two nodes to contain the same integer value.

We use two mechanically generated facts to ensure uniqueness of symbolic variables. To form these facts, we find all uses of symbolic variables (`SymbolicInt` and `SymbolicBool`). We describe the generation of facts for `SymbolicInt`. Similar facts are generated for `SymbolicBool`.

Consider a `sig A` where `B` is a field of type `SymbolicInt` – i.e. `B` is a relation of the type `A`→`SymbolicInt`. We form a list of all such relations {(`A1`, `B1`), (`A2`, `B2`), (`A3`, `B3`), ...} and then generate two facts.

The first fact ensures that all `SymbolicInt` atoms are used in one of these relations and the second fact ensures that we exactly have as many `SymbolicInt` atoms as needed in these relations. If any `SymbolicInt` atom is used in two relations, then some `SymbolicInt` atom is not used in any relation (because of second fact), but unused `SymbolicInt` atoms are not allowed (because of first fact). Thus the two facts are enough to ensure unique symbolic variables.

```
SymbolicInt = A1.B1 + A2.B2 + A3.B3 + ...
#SymbolicInt = #A1 + #A2 + #A3 + ...
```

Note that if some `sig` has more than one `SymbolicInt`, then for some i, j, `Ai = Aj`. The particular `sig` will be counted twice in the second fact. Also note that the new Alloy syntax requires the second fact to be written using the `plus` function as the + operator is dedicated to set union operation.

```
#SymbolicInt = (#A1).plus[(#A2).plus[(#A3).plus[ ... ]]]
```

3.4 Alloy Analyzer usage

This section discusses a practical issue in analyzing a model that contains symbolic clauses instead of concrete integers. The key problem is to deal with redundant clauses that may exist in a symbolic instance because they are allowed by the chosen scope, although not explicitly enforced by the constraints, i.e., to separate redundant clauses from enforced clauses. Recall that the Alloy Analyzer finds valid instances of the given model for the given scope. Any instance with redundant clauses within given bounds is still valid. These redundant clauses are not bound to any particular condition on the symbolic variables and can take many possible values resulting in the Alloy Analyzer showing many instances that are only different in the values of redundant clauses. We present two approaches to address this problem.

Iterative deepening The first approach is to iteratively run the Alloy Analyzer on increasing scopes for `Clause` atoms until we find a solution. The predicates in `symbolic` module require certain `Clause` atom to exist. If the scope for `sig Clause` is smaller than the number of required clauses, then the Alloy Analyzer will declare that no solutions can be found. This separate bound on `sig Clause` can be given as:

```
run RepOk for 3 but 1 Clause
```

There are three considerations in this approach. The first is performance. Performance is an issue for large models where the bound on `Clause` has to be tested from zero to some larger bound. However, for most models, Alloy analysis is often performed for small sizes. Thus the repetitions required for testing different values is also expected to be small. Still, this incurs a performance overhead.

The second consideration is how to decide an upper bound on number of clauses. The user may use multiple clauses on each symbolic variable. We can enumerate to twice the number of symbolic variables as a safe bound and then inform the user that there may be instances with more clauses but none with fewer clauses. If the user knows that their model needs more clauses, then they can give a higher bound for the clauses to find such instances.

The third consideration is if we find a solution with n clauses, there may be solutions with more than n clauses. For example, the user can write a predicate like:

```
a.eq[b] || (a.eq[c] && c.eq[b])
```

Such an expression can result in one to three clauses. If Alloy Analyzer finds a solution with n clauses, there might be solutions with $n+1$ and $n+2$ clauses. Because of this, when we find a valid solution, we inform the user that there might be solutions with more clauses. Again, the user – with knowledge of the model – can force a higher bound on clauses or rewrite such predicates.

Skolemization The second approach for handling the bound on `Clause` atoms uses *skolemization* in Alloy. According to Alloy's quick guide, "Often times, quantified formulas can be reduced to equivalent formulas without the use of quantifiers. This reduction is called *skolemization* and is based on the introduction of one or more skolem constants or functions that capture the constraint of the quantified formula in their values."

The important aspect of skolemization for our purpose is that skolemized atoms are identified explicitly in Alloy Analyzer's output. If we ensure that all generated clauses are skolemized we can start with a large bound for `Clause` atoms and easily identify redundant `Clause` atoms in the output.

Additionally, Alloy Analyzer's code can be modified to generate only skolemized atoms of one kind. This eliminates all issues related with bounds on the number of clauses. Only enforced clauses will be generated.

The only drawback to this scheme is that the user needs to ensure all predicates can be converted by skolemization. For example, the ordering check for sorted list in Section 2.3 does not produce skolemized results the way it is written. However the following equivalent predicate does:

```
some tail: l.header.*nextNode | no tail.nextNode
  && all n: l.header.*nextNode-tail | (n.data).lt[n.nextNode.data]
```

Instead of an implication, we have to use universal and existential quantifiers. The new sorting check for linked list works with skolemization.

Skolemization translates existential quantifier based expressions. In the future, it should be investigated if the technique associated with skolemization – that renames an atom generated to satisfy a predicate – can be separately used for symbolic execution of Alloy. This would require changing the Alloy Analyzer implementation and only allowing `Clause` atoms that are generated to satisfy predicates in the `symbolic` module. Such `Clause` atoms would be generated regardless of how the predicate in `symbolic` module was invoked.

4 Case Studies

This section presents four small case studies that demonstrate that our technique enables novel forms of analysis of Alloy models using the Alloy Analyzer.

4.1 Red-Black Trees

Red-black trees [6] are binary search trees with one extra bit of information per node: its color, which can be either red or black. By restricting the way nodes are colored on a path from the root to a leaf, red-black trees ensure that the tree is balanced, i.e., guarantee that basic dynamic set operations on a red-black tree take O(lg n) time in the worst case.

A binary search tree is a red-black tree if:

1. Every node is either red or black.
2. Every leaf (NIL) is black.

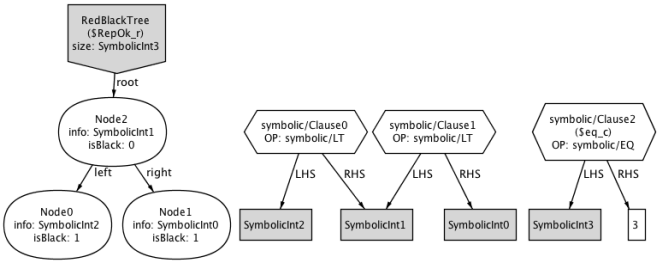


Fig. 2. Visualizing the constraints on data in a red-black tree with three nodes.

3. If a node is red, then both its children are black.
4. Every path from the root node to a descendant leaf contains the same number of black nodes.

All four of these red-black properties are expressible in Alloy [13]. Each node is modeled as:

```
sig Node {
  left: Node,
  right: Node,
  data: SymbolicInt,
  isBlack: Bool }
```

The core binary tree properties are:

```
pred isBinaryTree(r: RedBlackTree) {
  all n: r.root.*(left + right) {
    n !in n.^(left + right)    -- no directed cycle
    lone n.~(left + right)     -- at most one parent
    no n.left & n.right }}     -- distinct children
```

We show how symbolic execution of Alloy formulas helps in generating and visualizing red-black tree instances. Using symbolic execution for `size` is similar to sorted linked list. We now show how to make `data` symbolic and write the binary search tree ordering constraints using predicates in the `symbolic` module.

```
pred isOrdered(r: RedBlackTree) {
  all n: r.root.*(left+right) { -- ordering constraint
    some n.left => (n.left.info).lt[n.info]
    some n.right => (n.info).lt[n.right.info] }}
```

Next, we consider the `isBlack` relation. The constraints to validate color are:

```
pred isColorOk(r: RedBlackTree) {
  all e: root.*(left + right) | -- red nodes have black children
    e.isBlack = false && some e.left + e.right =>
      (e.left + e.right).isBlack = true
```

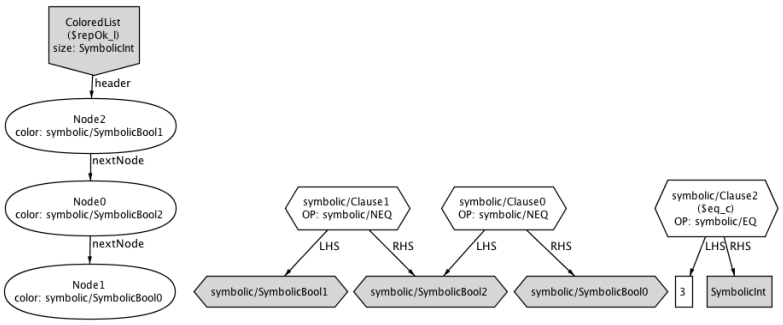


Fig. 3. Visualizing the constraints on a list with alternating colors. Presents an example with symbolic booleans.

```
  all e1, e2: root.*(left + right) | --all paths have same #blacks
    (no e1.left || no e1.right) && (no e2.left || no e2.right) =>
      #{ p: root.*(left+right) |
        e1 in p.*(left+right) && p.isBlack = true } =
      #{ p: root.*(left+right) |
        e2 in p.*(left+right) && p.isBlack = true }
}
```

We don't want `isBlack` to be symbolic because `isBlack` ensures that the generated trees are balanced. If we allow `isBlack` to be symbolic, the Alloy Analyzer will give instances with unbalanced trees combined with a set of unsolvable constraints for `isBlack`. To avoid such instances we keep `isBlack` concrete.

In Fig. 2, an example of a red-black tree instance produced by symbolic execution of the above model is shown. The `root` node is red while both children are black. The constraints show that `data` in `left` node has to be less than data in `root` node which has to be less than data in the `right` node. Another constraint shows that `size` has to be three for this red-black tree.

4.2 Colored List

In this example, we consider a list where no two successive elements have the same color. This example presents a case where symbolic booleans are used.

The `Node sig` is defined as:

```
sig Node {
 nextNode: lone Node,
 color: SymbolicBool }
```

The check for alternate colors in the list can be written as

```
pred ColorsOk(l: ColoredList) {
  all n: l.header.*nextNode |
    some n.nextNode => (n.color).neq[n.nextNode.color] }
```

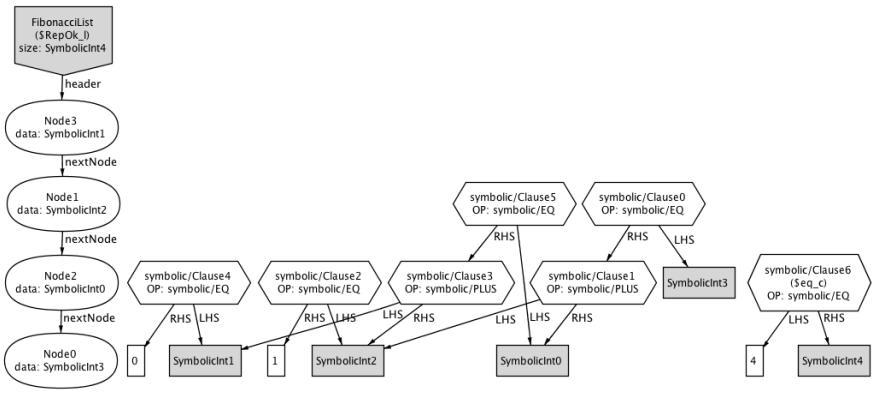


Fig. 4. Visualizing the constraints on data in a fibonacci sequence. Presents an example of non-trivial numeric constraints.

When this Alloy model is symbolically executed, one instance we get is shown in Fig. 3. There are expressions that restrict the value of each boolean to be *not equal* to either its predecessor's data or its successor's data.

Such a models help in visualizing the structure of a model and understanding the relationships between various elements. Since each symbolic instance corresponds to a class of concrete instances, we are able to visualize more structures and build a better understanding of the model in much less time.

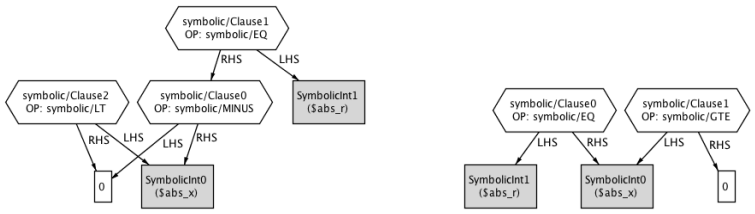
4.3 Fibonacci Series

This example presents how symbolic execution of Alloy models is able to allow non-trivial numeric operations and help avoid integer overflow. Because of Alloy's SAT-based analysis, the domain of integers used has to be kept small and integer overflow is a well-recognized issue. The Alloy 4.2 release candidate supports an option that disables generation of instances that have numeric overflow. Our approach provides an alternative solution since we build constraints on symbolic fields and do not require SAT to perform arithmetic.

This example considers a fibonacci series stored in a linked list. The first two elements are required to contain zero and one. Anything after that contains the sum of last two elements. This can be modeled in Alloy as:

```
pred isFibonacci(l: SortedList) {
  some l.header => (l.header.data).eq[0]
  some l.header.nextNode => (l.header.nextNode.data).eq[1]
  all n: l.header.*nextNode |
    let p = n.nextNode, q = p.nextNode |
      some q => (n.data).plus[p.data] &&
      (q.data).eq[(n.data).plus[p.data]] }
```

The first two constraints ensure that if the `header` and its `next` exist, they should be equal to 0 and 1 respectively. The third constraint works on all nodes



(a) First Branch (b) Second Branch

Fig. 5. Visualizing constraints on two paths within a small imperative function. Presents an example of visualizing traditional path conditions using Alloy.

(`n`) thats that have two more nodes (`p` and `q`) in front of them. It generates a `plus` clause between `n` and `p` and then generates an equality clause between the `plus` clause and `q`. This covers all restrictions on data in a fibonacci series.

Fig. 4 shows an instance of the fibonacci list with four nodes. The conditions show that the third and fourth node have to contain the sum of the previous two, while the first two nodes can only contain 0 and 1. This shows the expressive power of symbolic execution for Alloy models and the way it shows a whole class of concrete inputs in a single visualization.

4.4 Traditional symbolic execution of imperative code

This section demonstrates an example of a small imperative function that is translated to Alloy and is symbolically executed using the Alloy Analyzer. This shows a non-conventional application of the Alloy Analyzer. Consider the `abs` function from Section 2.1 that returns the absolute value of its input.

```
static int abs(int x) {
  int result;
  if (x < 0)
    result = 0 - x;
  else result = x;
  return result;      }
```

This function can modeled in Alloy as:

```
pred abs(x: Int, result: Int) {
  x.lt[0] => 0.minus[x] && result.eq[0.minus[x]]
    else x.gte[0] && result.eq[x] }
```

The predicate takes `x` and `result` where `x` is the original input and `result` models the return value of this function. Symbolic execution of this function explores two paths with conditions `x<0` on one path and `x>=0` on the other path.

When we run this model using symbolic execution for Alloy models, we find both these paths in the output of Alloy Analyzer. The visualization of these paths is shown in Fig. 5. Within the correct bounds and when redundant clauses are prevented, these are the only two results generated by the Alloy Analyzer.

This case study is one of the novel applications of symbolic execution in Alloy. It shows that Alloy can even provide a symbolic execution engine for traditional symbolic execution. It is yet to be seen how feasible Alloy would be in comparison with other symbolic execution engines for analysis of imperative programs.

5 Related Work

Clarke [5] and King [14] pioneered traditional symbolic execution for imperative programs with primitive types. Much progress has been made on symbolic execution during the last decade. PREfix [1] is among the first systems to show the bug finding ability of symbolic execution on real code. Generalized symbolic execution [12] shows how to apply traditional symbolic execution to object-oriented code and uses *lazy initialization* to handle pointer aliasing.

Symbolic execution guided by concrete inputs is one of the most studied approaches for systematic bug finding during the last five years. DART [10] combines concrete and symbolic execution to collect the branch conditions along the execution path. DART negates the last branch condition to construct a new path condition that can drive the function to execute on another path. DART focuses only on path conditions involving integers. To overcome the path explosion in large programs, SMART [9] introduced inter-procedural static analysis techniques to reduce the paths to be explored by DART. CUTE [15] extends DART to handle constraints on references. CUTE can in principle be used with preconditions on structural inputs.

EGT [2] and EXE [3] also use the negation of branch predicates and symbolic execution to generate test cases. They increase the precision of the symbolic pointer analysis to handle pointer arithmetic and bit-level memory locations. All the above approaches consider symbolic execution for imperative constraints.

Symbolic Execution has been applied outside the domain of imperative programs. Thums and Balser [16] uses symbolic execution to verify temporal logic and statecharts. They consider every possible transition and maintain the symbolic state. Wang et al. [18] use symbolic execution to analyze behavioral requirements represented as Live Sequence Charts (LSC). LSC are executable specifications that allow the designer to work out aberrant scenarios. Symbolic execution allows them to group a number of concrete scenarios that only differ in the value of some variable. These are novel applications of symbolic execution, however, they translate the problem from some domain to a sequence of events with choices. This is essentially a sequential operation. To our knowledge symbolic execution has not yet been applied to declarative logic programs.

The Alloy Analyzer uses the Kodkod tool [17], which provides the interface to SAT. The Alloy tool-set also includes JForge [8], which is a framework for analyzing a Java procedure against strong specifications within given bounds. It uses Kodkod for its analysis. JForge translates an imperative Java program to its declarative equivalent. We believe JForge can provide an enabling technology to transform our technique for symbolic execution of Alloy models to handle imperative programs.

6 Conclusion

This paper introduced a novel technique for symbolic execution of declarative models written in the Alloy language. Our insight is that the fully automatic SAT-based analysis that the Alloy tool-set supports, provides a form of execution that can be leveraged to define symbolic execution for Alloy. We demonstrated the efficacy of our technique using a variety of small but complex Alloy models, including red-black trees, colored lists, Fibonacci series, as well as a model of an imperative program. We believe our work opens exciting opportunities to develop more efficient and effective analyses of Alloy. For example, the constraints on symbolic fields can be solved using specialized solvers that directly support the corresponding theories. Moreover, a symbolic instance summarizes a number of concrete instances and thus our technique provides an efficient mechanism for the user to enumerate and inspect desired instances.

Acknowledgments

This work was funded in part by the Fulbright Program, the NSF under Grant Nos. IIS-0438967 and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

1. Bush, W.R., et al.: A Static Analyzer for Finding Dynamic Programming Errors. Softw. Pract. Exper. **30**(7) (2000)
2. Cadar, C., Engler, D.: Execution Generated Test Cases: How to make systems code crash itself. In: SPIN 2005
3. Cadar, C., et al.: EXE: Automatically Generating Inputs of Death. In: CCS 2006
4. Cadar, C., et al.: Symbolic Execution for Software Testing in Practice – Preliminary Assessment. In: ICSE Impact 2011
5. Clarke, L.A.: Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation. PhD thesis, University of Colorado at Boulder (1976)
6. Cormen, T.T., et al.: Introduction to Algorithms. MIT Press (1990)
7. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS 2008
8. Dennis, G., Yessenov, K.: Forge website. `http://sdg.csail.mit.edu/forge/`
9. Godefroid, P.: Compositional Dynamic Test Generation. In: POPL 2007
10. Godefroid, P., et al.: DART: Directed Automated Random Testing. In: PLDI 2005
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
12. Khurshid, S., et al.: Generalized Symbolic Execution for Model Checking and Testing. In: TACAS 2003
13. Khurshid, S., Marinov, D.: TestEra: Specification-Based Testing of Java Programs using SAT. Automated Softw. Eng. J. **11**(4) (2004)
14. King, J.C.: Symbolic Execution and Program Testing. Commun. ACM **19**(7) (1976)
15. Sen, K., et al.: CUTE: A Concolic Unit Testing Engine for C. In: ESEC/FSE 2005
16. Thums, A., Balser, M.: Interactive Verification of Statecharts. In: INT 2004
17. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: TACAS 2007
18. Wang, T., et al.: Symbolic Execution of Behavioral Requirements. In: Pract. Aspects Decl. Lang. (2004)