Pythia: Automatic Generation of Counterexamples for ACL2 using Alloy

Alexander Spiridonov*
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
alspirid@microsoft.com

Sarfraz Khurshid
The University of Texas at Austin
1 University Station C5000
Austin, TX 78712
khurshid@ece.utexas.edu

ABSTRACT

A key research problem in automated theorem proving is generating examples and counterexamples to guide the discovery of proofs. We present Pythia, a framework that connects ACL2 with the SAT-based Alloy Analyzer, a tool for solving formulas in first-order logic with transitive closure using bounded exhaustive checking. Pythia takes as inputs an Alloy model of the ACL2 type system together with an ACL2 formula and automatically generates examples of ACL2 objects that satisfy the model's constraints. Pythia then produces an ACL2 script that evaluates the formula on the generated objects to search for counterexamples. We test Pythia on a set of classic ACL2 non-theorems and find that it effectively discovers counterexamples to such formulas. Based on our experiments, we suggest making ACL2 more novice-friendly by adding the option to test every formula on a set of basic examples before attempting the proof.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods*

General Terms

Verification

Keywords

ACL2, Alloy, SAT, counterexamples

1. INTRODUCTION

While substantial progress has been made in automating theorem provers, constructing a complex proof remains a challenging task, particularly for novice users. When ACL2 fails to prove a given formula, there are two possible explanations: the formula is not a theorem (i.e., valid under all assignments of values to the formula's free variables); or the formula is indeed a theorem, but the user lacks some insight, such as a key lemma, that would enable the proof to succeed. Unfortunately, at present ACL2 does not distinguish between these two reasons for failure, which can frustrate the novice user. Furthermore, when the formula is not a theorem, a counterexample (i.e., an assignment to the free variables of the formula that makes the formula false) would demonstrate exactly when the formula is false and may suggest a way to turn it into a theorem. However, novice users might find manually constructing counterexamples difficult. Thus, a key research problem in automated theorem proving is automatically generating examples and counterexamples to guide the discovery of proofs [5]. An automated tool could alleviate the burden on novice users and help them interact with ACL2 more effectively.

We present Pythia, a framework that connects ACL2 with the SAT-based Alloy Analyzer [3], a tool for solving formulas in first-order logic with transitive closure using bounded exhaustive checking. The Alloy Analyzer takes a complementary approach to theorem proving: instead of trying to prove a formula, the Analyzer looks for a counterexample. The Analyzer leverages off-the-shelf SAT technology to perform bounded exhaustive checking using small bounds on the universe of discourse. By integrating ACL2 with Alloy, Pythia has immediate practical use: before trying to prove a formula using ACL2, the user can first check whether there are any small counterexamples using the Alloy Analyzer. For formulas that have small counterexamples, e.g., formulas often written by novice users, such an approach can save a lot of frustration.

Pythia takes as inputs a model of the ACL2 type system written in the Alloy specification language [2] and an ACL2 formula. Our model represents the built-in ACL2 types in Alloy: the complex rationals (including the rationals, integers, and naturals), characters, symbols, strings, and cons pairs. While our model is inevitably a simplification of ACL2, it enables Pythia to leverage Alloy without modeling every detail of the ACL2 logic in Alloy. Pythia invokes the Alloy Analyzer to automatically generate examples of ACL2 objects that satisfy the model's constraints. Pythia then translates the generated objects from Alloy to their ACL2 representations and produces an ACL2 script that evaluates the formula on these objects to search for counterexamples. Whenever an assignment makes the formula false, ACL2 outputs that assignment as a counterexample.

*This research was done while the author was a graduate student in the Department of Computer Sciences at The University of Texas at Austin.

Pythia uses Alloy to generate examples of Alloy constraints. The Alloy constraints need not — indeed, usually do not — capture all the constraints present in the ACL2 formula in question. In fact, in its initial state, the Pythia constraints consist entirely of a model of the ACL2 type system and Pythia simply enumerates in a random order examples of objects of the various types, without any sensitivity to the larger ACL2 context (which is unavailable to the Alloy constraint solver). These examples are then tried in ACL2. Experiments show that in some simple, classical non-theorems, these examples are sufficient. This leads us to make a suggestion to the ACL2 developers:

Why not test every formula before beginning proof by evaluating it on several dozen "typical" examples? If a formula evaluates to false on any of these examples, ACL2 could display that example together with a descriptive error message, instead of the traditional (and rather terse) "The proof attempt has failed." You might thereby save the novice a lot of heartburn!

Pythia takes the first step towards leveraging Alloy to search for examples and counterexamples. To get the full advantage of the Alloy constraint solver, at present one must manually translate the ACL2 constraints from the problem into Alloy. This can be difficult since Alloy does not support general recursive schemes. In a subsequent section we provide an example of extending our basic model with a more complex recursive ACL2 predicate — an ordered list of integers — and propose some general strategies for translating recursive ACL2 definitions into Alloy.

Our paper makes the following contributions:

- **Connection with Alloy** We take the first step towards connecting ACL2 with Alloy, which is a powerful and widely used tool in the formal methods community. We present an Alloy model of the ACL2 type system that the Alloy Analyzer can use to automatically generate examples illustrating ACL2 definitions.
- **Automatic discovery of counterexamples** Pythia automatically translates the generated examples from Alloy to ACL2 and uses them to search for counterexamples to the provided formula. Our experiments show that Pythia successfully uncovers counterexamples to the kinds of non-theorems a novice user is likely to encounter.
- **A suggestion for improving ACL2** Because even a quick search for counterexamples could improve the experience for novice users, we propose making ACL2 more novice-friendly by adding the option to test proposed conjectures on a set of basic examples before attempting the proof. Pythia provides one possible implementation of this improvement.

The rest of this paper is organized as follows. Section 2 provides the relevant background on Alloy. Section 3 describes the Pythia framework, our Alloy model of ACL2 types, and ways to model recursive ACL2 definitions in Alloy. Section 4 presents our experimental results, which demonstrate Pythia's effectiveness in generating counterexamples to classic ACL2 non-theorems. Section 5 discusses current limitations of our work and directions for future research, and Section 6 concludes.

2. ALLOY

The Alloy toolset [2] consists of a specification language and an analyzer that automatically generates instances of models written in that language.

2.1 The Specification Language

The Alloy specification language is a strongly-typed, first-order declarative language based on sets and relations. An Alloy model (specification) contains *signatures*, which define the objects in the model and the relations between them, and *constraint paragraphs*, which record various constraints and expressions.

A *signature* introduces a set of atoms. The declaration

```
sig Node {
  left, right : lone Node,
  value : lone Int
}
```

introduces a set named `Node`. We use this signature to define a rooted binary tree:

```
sig Tree {
  root : Node
}
```

Relations are declared as fields of signatures. The `Node` declaration introduces binary relations `left`, `right`, and `value`. The relations `left` and `right` map a node in a binary tree to its left and right children, while `value` maps a leaf node to an integer value.

Relations can be combined to form expressions. The set operators union (`+`), intersection (`&`), and difference (`-`) combine two relations of the same type. The dot operation (`.`) represents relational composition. Alloy also supports the unary operators `~` (transpose), `^` (transitive closure), and * (reflexive transitive closure).

Expression quantifiers turn expressions into formulas. The formula `no e` is true when `e` denotes a relation containing no tuples. Similarly, `some e`, `lone e`, and `one e` are true when `e` has some, at most one, and exactly one tuple respectively. Formulas can be made with relational comparison operators: subset (written `:` or `in`), equality (`=`) and their negations (`!:`, `!in`, `!=`). For example, `e1 in e2` is true when every tuple in (the relation denoted by the expression) `e1` is also a tuple of `e2`. Formulas can be combined with the standard logical operators: `&&` (conjunction), || (disjunction), `=>` (implication), `<=>` (bi-implication), and `!` (negation). A sequence of formulas within curly braces is implicitly conjoined.

A *declaration* is a formula `v op e` consisting of a variable `v`, a comparison operator `op`, and an arbitrary expression `e`.

Quantified formulas consist of a quantifier, a comma separated list of declarations, and a formula. In addition to the universal and existential quantifiers `all` and `some`, there is `lone` (at most one) and `one` (exactly one). In a declaration, `part` specifies partition and `disj` specifies disjointness.

Constraint paragraphs include facts and predicates. A *fact* is a constraint that is assumed always to hold. A *predicate* is a named constraint, with zero or more declarations for arguments. When a predicate is used, an expression must be provided for each argument; its meaning is the predicate's constraint with each argument replaced by its instantiating expression. For example, the following predicate places an acyclicity constraint on its argument, which is of type `Tree`:

```
pred Acyclic (t: Tree) {
  all n : t.root.*(left+right) |
    n !in n.^(left+right)
}
```

The expression `t.root.*next` denotes the set of all nodes reachable from the root node of the tree following zero or more traversals along the left and right branches; similarly, the expression `n.^(left+right)` denotes the set using one or more traversals. This predicate thus states that for all nodes that are reachable from the root of the tree, traversing from such a node along the left or right pointer one or more times does not lead back to the same node.

2.2 The Analyzer

The Alloy Analyzer is an automatic tool that finds *instances* of Alloy specifications. An instance assigns values to the sets and relations in the specification such that all formulas in the specification evaluate to true. Because first-order logic is undecidable, the Analyzer limits its search to a finite scope, i.e. a bound on the universe of discourse. The analysis is performed by translating the model into a Boolean formula and using a SAT solver to find an assignment satisfying the formula.

Given the predicate `Acyclic` defined above, we can write a *command* that will tell the Analyzer to find an instance of the model that satisfies the predicate's constraint:

```
run Acyclic for 3 but 1 Tree
```

This command sets the scope for analysis: it limits the search to a universe in which each top-level set has at most 3 elements, but the `Tree` set has at most 1 element. Effectively, this command will generate a single acyclic tree with up to 3 nodes.

In the next section we describe how we leverage the Alloy toolset to automatically generate ACL2 objects and how we use these objects to search for counterexamples to ACL2 formulas.

3. PYTHIA

Pythia is a framework for finding examples and counterexamples for ACL2 definitions and formulas. Pythia uses the Alloy Analyzer to generate instances of ACL2 objects and ACL2 to evaluate the formula on these instances and report counterexamples. Pythia is written in Java and uses the Alloy API to interact with the Alloy Analyzer.

3.1 Architecture

Pythia takes two inputs: an Alloy model of the ACL2 type system and an ACL2 formula. Pythia's operation proceeds in two phases.

In the first phase, Pythia invokes the Alloy Analyzer to generate an instance of the input Alloy model. The resulting instance is an Alloy representation of ACL2 objects that satisfy the model's constraints. By default, the Analyzer generates up to three examples of each atomic ACL2 type and two cons trees: a proper cons tree and an improper cons tree. The user may override this default and specify how many objects will be generated and what properties these objects will have by listing the predicates the model must satisfy and by setting the scope for analysis. Lastly, the user may indicate the type of values that will appear in the generated cons trees. This may be a single type (such as integer or string) or a combination of types (for example, characters and symbols.) By default, the generated cons trees may contain objects of any atomic ACL2 type.

In the second phase, Pythia translates the generated objects from their Alloy representations into ACL2, creating a vector of test inputs. Pythia outputs an ACL2 file containing the original formula and some test code. When the user loads the produced file in ACL2, ACL2 will execute the test code, evaluating the formula on all combinations of assignments of values from the test vector to the formula's variables. Whenever the formula evaluates to false, ACL2 outputs the current assignment as a counterexample to the formula.

3.2 Modeling ACL2 Types in Alloy

Pythia requires a model of the ACL2 types from which the Alloy Analyzer will generate examples of ACL2 objects. We developed a model that captures all the built-in ACL2 types. The user may extend the model with additional constraints in first-order logic so that objects with particular properties may be generated: for example, proper cons trees whose elements are integers, appearing in ascending order. One advantage of our model is its modularity: our predicates define building blocks that can be combined to introduce more complex constraints. We first describe the modeling of atomic ACL2 types and then discuss the representation of cons trees.

3.2.1 Modeling Atomic ACL2 Types

We begin by modeling integers as subsets of the Alloy `Int` type:

```
sig PositiveInt extends Int {}
sig NegativeInt extends Int {}
one sig Zero extends Int {}

fact { all p : PositiveInt | int p > 0 }
fact { all n : NegativeInt | int n < 0 }
fact { int Zero = 0 }
```

Next, we model ratios. The signature `Ratio` is *abstract*: it has no elements except those belonging to its subsets, `PositiveRatio` and `NegativeRatio`.

```
abstract sig Ratio {
  numerator, denominator: Int
}

sig PositiveRatio extends Ratio {}
sig NegativeRatio extends Ratio {}
```

We specify additional constraints on each type of ratio:

```
fact { all p : PositiveRatio |
  int p.numerator > 0 && int p.denominator > 0 ||
  int p.numerator < 0 && int p.denominator < 0}
fact { all n : NegativeRatio |
  int n.numerator < 0 && int n.denominator > 0 ||
  int n.numerator > 0 && int n.denominator < 0}
```

We then use ratios to model complex rationals, with the constraint that the imaginary parts must be non-zero:

```
sig ComplexRational {
  real, imaginary: Ratio
}

fact {
  all c : ComplexRational |
    int c.imaginary.numerator != 0
}
```

Finally, we model symbols, strings, and characters as simple Alloy signatures:

```
sig Symbol {}
one sig T extends Symbol {}
one sig NIL extends Symbol {}
sig String {}
sig Character {}
```

We keep our model of these types simple; Pythia does all the work of translating these objects into their valid ACL2 representations.

3.2.2 Modeling Cons Trees

We model cons trees as rooted binary trees, using the signatures `Node` and `Tree` introduced in Section 2:

```
module Node [t]

sig Node {
  left, right: lone Node,
  value : lone t
}

sig Tree {
  root: Node
}
```

The `Node` signature is *parameterized*: it resides in a separate module, and the main model imports this module, instantiating it with the name of a signature. Parameterization is a syntactic mechanism; the effect of instantiating a `Node` module is the same as adding a copy of the module with the parameter `t` replaced by a name of some signature. This allows the user to control the types of values appearing in the cons trees; our model enables the generation of cons trees whose elements are any single ACL2 type or any combination of ACL2 types. Each node has at most one left and right child and an optional value; a node with no children represents `nil` (the empty list.)

Having defined a binary tree, we introduce a predicate `Cons` that constrains a tree to be a valid cons tree:

```
pred Cons(t: Tree) {
  all n : t.root.*(left+right) |
    n !in n.^(left+right)
  all n : t.root.*(left+right) |
    lone n.~(left+right)
  all n : t.root.*(left+right) |
    some n.(left+right) => no n.value
  all n : t.root.*(left+right) |
    no n.(left+right) or #n.(left+right) = 2
}
```

The predicate states that a cons tree is acyclic; each node has at most one parent; nodes with children (i.e. non-leaf nodes) do not have values; and each node has either two children or none (cons trees are, by definition, full binary trees.)

Because these properties are intrinsic to all cons trees, we can use the `Cons` predicate as a building block to specify cons trees with additional properties, such as a proper (`nil`-terminated) cons tree:

```
pred ProperCons(t: Tree) {
  Cons(t)
  one n : t.root.^right |
    no n.(left+right) && no n.value
}
```

A proper cons tree, in addition to having all the properties of a cons tree, has one node in the right sub-tree that has no children and no value; in effect, the right-most branch in the tree terminates in `nil`. We model an improper cons tree similarly. We model a true list of atoms, a common type of cons tree, by reusing the `ProperCons` predicate and adding one more constraint:

```
pred TrueListOfAtoms(t: Tree) {
  ProperCons(t)
  all n : t.root.*(left+right) |
    some n.(left+right) =>
      one n.left.value && no n.right.value
}
```

A true list of atoms, in addition to having all the properties of a proper cons tree, has the following property: for all of

its nodes, the left child must have a value, and the right child must not have a value (thus it is either a non-leaf node or `nil`.) An improper list of atoms is defined analogously.

Lastly, we model a true list of integers whose values are ordered. First, we define a predicate for an ordered tree: one in which values in the left subtree are smaller than all the values in the right subtree.

```
pred Ordered(t: Tree) {
  all n : t.root.*(left+right) |
    all v : n.left.*(left+right).value |
      all w : n.right.*(left+right).value |
        int v < int w
}
```

We combine this predicate with the definition of `TrueListOfAtoms` to define an ordered true list of integers:

```
pred OrderedTrueListOfIntegers(t: Tree) {
  Ordered(t)
  TrueListOfAtoms(t)
}
```

This predicate corresponds to the following ACL2 definition:

```
(defun ordered-true-list-of-integers (x)
  (and (integer-listp x)
       (orderedp x)))
```

Running the Alloy Analyzer on our model generates an instance that contains an object satisfying the `OrderedTrueListOfIntegers` predicate. After obtaining the instance from the Analyzer, Pythia automatically translates the object into its ACL2 representation. This object illustrates the definition `ordered-true-list-of-integers` and becomes one of the entries in the test vector that is then used to find counterexamples to ACL2 formulas.

3.3 Modeling Recursive Functions in Alloy

Recursive functions are very common in ACL2; unfortunately, Alloy does not support recursive definitions. We describe three ways of overcoming this problem: unrolling recursive functions to create non-recursive constraints; using transitive closure; and introducing a new relation to compute the fixed point of the recursive function.

The first approach to translating a recursive ACL2 function to Alloy is to unroll the recursion a fixed number of steps. Consider the following ACL2 definition of a predicate to recognize an ordered list:

```
(defun orderedp (x)
  (cond ((atom x) t)
        ((atom (cdr x)) t)
        (t (and (<= (car x) (cadr x))
                (orderedp (cdr x))))))
```

This definition states an atom is "ordered"; a singleton list is ordered; and if the list has more than one element, it is ordered if the first element is less than or equal to the second element, and the rest of the list is ordered. While we cannot directly translate this definition into Alloy, we can unroll it, generating the following set of constraints:

```
(and (consp x)
     (consp (cdr x))
     (< (car x) (cadr x))
     (null (cddr x)))
```

We can then formulate an Alloy predicate that captures these constraints:

```
pred Orderedp(t: Tree) {
  Cons(t) //basic "cons" constraints
  t.left.value < t.left.right.value
  no t.right.right.value // right-most node is NIL
}
```

The second method is to model a recursive ACL2 definition with an Alloy predicate that uses transitive closure. Consider the ACL2 function to check membership in a cons tree:

```
(defun mem (e x)
  (if (endp x)
      nil
    (or (equal e (car x))
        (mem e (cdr x)))))
```

The recursion in `mem` can be eliminated in favor of transitive closure:

```
pred mem(x: Tree, e: Elem) {
  e in lst.*(left+right).value
}
```

Lastly, it is possible to eliminate recursion by introducing a new relation that computes the fixed point of the recursive function [6]. Here is the definition of a function `len` that finds the length of a list:

```
(defun len (x)
  (if (endp x)
      0
    (+ 1 (len (cdr x)))))
```

This definition can be modeled in Alloy as follows:

```
one sig Recursion {
  len: Node -> Int
}

fun lenFix(n: Node): Int {
  no n.right => Int[0]
```

```
  else Int[int[Recursion.len[n.right]] + 1]
}

fact FixPoint {
  all n: Node {
    Recursion.len[n] = lenFix[n]
  }
}
```

4. EXPERIMENTAL RESULTS

As an initial evaluation, we tested Pythia's ability to generate counterexamples for 15 ACL2 formulas that are not theorems. Most formulas came from [4] and represented typical conjectures that novice users try to prove. The formulas ranged from simple conjectures with no hypotheses to more complex statements whose hypotheses included recursive functions. Pythia successfully found counterexamples to all 15 formulas. Below we show some formulas and counterexamples that Pythia discovered.

Consider the following formula (the function `rev` is defined in the standard way, as in [4]):

```
(equal (rev (rev x)) x)
```

This formula attempts to capture the intuitive notion that reversing the same object twice produces the original object. While this statement is true when the object is a true list, it is not true for improper lists and atoms. The counterexamples that Pythia finds illustrate this fact:

```
x = (2 . 3)
x = #C(3 3)
x = String0
x = T
```

Next, consider a formula that tries to formalize the relationship between the functions `rev` and `app` (also defined in [4]):

```
(equal (rev (app a b))
       (app (rev a) (rev b)))
```

This formula contais an error: the calls to `rev` in the right-hand side of the equality should be swapped. Any pair of distinct objects could serve as a counterexample, and Pythia produces many such pairs, such as:

```
a = (2 . 3); b = (0 -1)
a = 'SYMBOL0; b = 'SYMBOL1
a = 'SYMBOL1; b = (2 . 3)
```

Finally, the following formula has a hypothesis containing calls to recursive functions `integer-listp` and `orderedp`:

```
(implies (and (integer-listp x)
              (integer-listp y)
              (orderedp x)
              (orderedp y))
         (orderedp (app x y))))
```

The predicate `OrderedTrueListOfAtoms`, defined in Section 2, models the combination of these recursive definitions, allowing Pythia to generate objects with the required properties and produce a counterexample:

```
x = (2 3); y = (-1 0 1)
```

5. DISCUSSION

Prior efforts to automatically find counterexamples to ACL2 formulas by using SAT include the work of Sumners [8], who presents a procedure for checking "suitably bounded" ACL2 theorems using a SAT checker that is also written in ACL2, and Reeber and Hunt [7], who define the subclass of unrollable list formulas in ACL2 (SULFA) and extend ACL2 to recognize SULFA formulas and automatically verify them with a SAT-based decision procedure.

Pythia takes a novel approach by connecting ACL2 with an external tool, the Alloy Analyzer. This connection allows ACL2 users to leverage Alloy's strength: the ability to solve complex conjunctions of constraints through bounded exhaustive checking. The immediate practical advantage is that before trying to prove a formula using ACL2, the user can first check for counterexamples using the Alloy Analyzer. For formulas that have small counterexamples, e.g., formulas often written by novice users, this approach can be effective and save a lot of frustration.

The connection between Alloy and ACL2 can also be used in the other direction: ACL2 could help Alloy perform bounded exhaustive checking. Consider checking a complex formula using the Alloy Analyzer. Suppose the translation of the formula generates a SAT problem that is infeasible to solve. The user could try to decompose the original formula into subformulas and check some of them using ACL2 in an attempt to simplify the original formula by reducing them to constants, similarly to the approach described in [1].

While Pythia takes the first step towards integrating ACL2 and Alloy, two limitations preclude it from being applied to formulas of greater complexity.

Pythia's first limitation is that although our approach is sound, it is not complete: Pythia may not find a counterexample to a formula even if one exists because none of the objects Pythia generates may turn out to be valid counterexamples. Two factors mitigate this drawback. First, by default we generate instances of all primitive ACL2 types, which yields coverage that is often sufficient to find counterexamples to simple theorems, as illustrated by our experimental results. Second, our model allows the user to add constraints and to specify the scope for analysis. A greater scope increases the chances of generating a counterexample, at the expense of increased analysis time.

The second limitation is that in its initial state, Pythia cannot handle recursive functions effectively and automatically. While our model does not require any user involvement to

generate examples of all primitive ACL2 types and cons trees with certain properties (such as ordered true lists of integers), at present the user must manually define additional constraints in Alloy in order to represent more complex definitions. This obstacle can be overcome by developing an automatic translator from ACL2 formulas to Alloy using the techniques for representing recursive definitions in Alloy described in Section 3.

The next step, therefore, is to implement translation from ACL2 formulas to Alloy formulas based on our Alloy model to provide a fully automatic tool that enables bounded exhaustive checking of a range of ACL2 formulas. Generating instances using the Alloy Analyzer and checking each of them against the ACL2 formula can require a large number of instances before a counterexample is found. A translation of the ACL2 formula to Alloy would enable SAT to directly look for a counterexample. Another future direction is to prove some standard Alloy models — particularly the ones for which the Analyzer does not find any counterexamples — using ACL2. Such a study would help us determine how often bounded exhaustive checking gives false confidence in the correctness of a formula and how useful the Analyzer is in building lemmas to construct a proof in ACL2.

6. CONCLUSIONS

Pythia is a novel framework that connects ACL2 with the SAT-based Alloy Analyzer in an effort to automatically find examples and counterexamples for ACL2 formulas. The Alloy Analyzer takes a complementary approach to theorem proving: instead of trying to prove a formula, the Analyzer looks for a counterexample. The Analyzer leverages off-the-shelf SAT technology to perform bounded exhaustive checking using small bounds on the universe of discourse. By integrating ACL2 with Alloy, Pythia has immediate practical use: before trying to prove a formula using ACL2, the user can first check whether there are any small counterexamples using the Alloy Analyzer. For formulas that have small counterexamples, e.g., formulas often written by novice users, such an approach can save a lot of frustration. This leads us to propose making ACL2 more novice-friendly by adding the option to test each formula on a set of basic examples before attempting the proof.

Pythia takes the first step towards integrating ACL2 with the Alloy Analyzer. In addition to automatically finding counterexamples, we envision other interesting applications of this collaboration, such as using ACL2 to help Alloy overcome SAT problems that are infeasible to solve directly. We hope that in the future, by working together and leveraging each other's strengths, ACL2 and Alloy could help both novices and experts achieve results that would be difficult or even impossible to obtain using either tool only by itself.

7. ACKNOWLEDGMENTS

We are grateful to J Strother Moore for his comments on an early draft of this paper.

8. REFERENCES

[1] K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating model checking and theorem proving for relational reasoning. In *Seventh International Seminar on Relational Methods in Computer Science (RelMiCS 2003)*, volume 3015 of *Lecture Notes in Computer Science (LNCS)*, pages 21–33, Malente, Germany, May 2003.

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

[3] D. Jackson, I. Scheckter, and I. Shlyakhter. Alcoa: The Alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 730–3, Limerick, Ireland, June 2000.

[4] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[5] M. Kaufmann and J. S. Moore. Some key research problems in automated theorem proving for hardware and software verification. In *RACSAM*, volume 98(1), pages 181–195. Spanish Royal Academy of Science, 2004.

[6] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 231–245, 2002.

[7] E. Reeber and W. A. H. Jr. A SAT-based decision procedure for the subclass of unrollable list formulas in ACL2 (SULFA). In *Automated Reasoning, Third International Joint Conference*, Aug. 2006.

[8] R. Sumners. Checking ACL2 theorems via SAT checking. In *Third International Workshop on the ACL2 Theorem Prover and Its Applications*, Feb. 2002.