Interactive (Non)Theorem (Dis)Proving

Harsh Raju Chamarthi

August 21, 2015

Abstract

We present a framework for interactively disproving non-theorems. Our method can be used to add automated disproving and counterexample generation capabilities to existing interactive theorem provers. This capability increases the utility and effectiveness of theorem provers because it provides automated support for what users spend most of their time doing: debugging flawed models, invariants, interfaces and conjectures. We present various disproving techniques and discuss an implementation and evaluation of the framework using ACL2s, the ACL2 Sedan.

1 Introduction

Formal, computer-aided reasoning is a fundamental and promising method for helping designers build complex and dependable systems. Interactive theorem provers[1] provide rich formalisms and powerful inference methods to assist users in this activity. However, the purpose of almost all existing methods is to aid in the construction of formal proofs, *i.e.*, given a true conjecture, these methods are designed to help the user find a proof. But what of the numerous initial false starts, faults in modeling, invalid conjectures, abstraction mistakes, etc. that overwhelmingly account for where users actually spend their time? We propose that methods for finding counterexamples and disproving non-theorems, warrant proportionally more support. In fact, mathematical reasoning often proceeds as a succession of these duals, proving and disproving, both informing each other in a synergistic interplay. What then of the dual of Interactive Theorem Proving? This line of reasoning leads us to an Interactive Non-Theorem Disprover! The capability to interact with the user to explicitly equip the tool in discovering, in addition to proofs, counterexamples, is either non-existent or severely lacking in the current state-of-art interactive theorem provers. In this report, we investigate and provide a framework to address this deficiency.

To arrive at a plausible method, we need to obtain a good working idea of the nature and form of the mathematical process of discovery, the activity we intend to assist. Exploration, experimentation and observation are preludes to discovery, be it the discovery of a pattern, a conjecture, an algorithm and its invariants, a simplifying abstraction, a proof strategy, etc. Concrete calculation on and testing of one's developing intuition using representative examples and special cases are very important methods of exploration. The experiments and the observations of results gleaned provide a clearer picture of the object under study and its properties, dispel or confirm user intuition and identify mistakes and gaps in the formalization. One might daresay it is by experimenting with simple examples and special cases, that their general properties are intuited; and boldly add that it is the failure to construct a counterexample to a perceived conjecture that convinces the user of its truth and, often, points to a line of proof. The importance of concrete construction and calculation cannot be underestimated, and our method mirrors (builds on) this observation.[2]

The basic ingredient of our method is this: for every symbolic/analytic characterization of the structure and form of the objects under consideration, we also arrange for its constructive/synthetic analogue, *i.e.*, in

[1] Also known as "Proof Assistants," *e.g.*, ACL2, Coq, HOL4, HOL-Light, Isabelle, NuPrl, PVS etc.

[2] Though this restricts it to executable (fragments of) logics/languages, it is widely applicable to the field of computer-aided reasoning.

addition to a predicate that recognizes objects, the user provides a generative characterization that produces objects of the desired form, and proves that it is sound (*i.e.*, generated objects satisfy their predicate).

In the case of monadic predicates, the user input usually takes the form of *enumerator* functions whose domain is $\mathbb{N}$, *e.g.*, `(nth-prime n)` denotes the nth number satisfying predicate `primep`. This situation reduces the sample space of an individual variable, down from the entire universe of discourse, to its type/sort domain. In fact the success of Quickcheck-like tools for property-based testing hinges on the use of such enumerators for type-like monadic constraints.

Unfortunately, the hypotheses (pre-conditions) in typical conjectures (properties) are more complex than a sequence of type restrictions. The search space is still huge, and randomly sampling the domain (restricted by type or monadic constraints) of each variable is still akin to searching for needles in bundles of large haystacks, rather than one huge haystack (universe). Can we search (for counterexamples) more efficiently? In the context of an interactive theorem prover (ITP), indeed there is ample scope. The ITP itself is equipped with an array of symbolic reasoning methods. In addition ITPs can be extended by using libraries that provide access to powerful theory reasoning, or by user-provided invariants and theorems, which allow the theorem prover to become adept at reasoning about the domain under consideration. All of this is brought to the fore, to magnify the help given by the user (who provides basic enumerative characterization), and combined to realize a more efficient counterexample search. We present a counterexample generation utility, Cgen, that synergistically combines property-based testing with the powerful proof-based search of the ITP itself [6, 9].

The success of interactive tools, crucially depends on ease of use, conferring value to user's time, effort and intelligence. We have paid special attention to the interface of user input characterizing the enumerative aspect of monadic predicates; if it is possible to mechanically generate the enumerator functions from the predicate definitions, we do so. In fact, we have designed a simple and intuitive language to describe data (objects), that automatically generates both its predicate and enumerator definitions, along with the necessary metadata required by Cgen [7]. The data definition framework (Defdata), implementing this language, also provides useful primitives for writing and associating custom enumerators for arbitrary monadic predicates. Moreover, the defdata facility generates many theorems automating type-based reasoning; in particular, it records subtype and disjoint type relations that Cgen exploits.

The ITP's inference methods, proof-search heuristics and the model-specific theorems (lemmas) provided by users (directing the ITP to the final proof) can vastly simplify the proof of conjectures. In concert with property-based testing at the subgoals (leaves) of the proof search, this goes a long way in aiding the search for counterexamples. Nevertheless, there are cases in which it falls short.

The simplified hypotheses (of the subgoals) are often more complex than a sequence of independent monadic constraints enumerable by Cgen; a variable might be constrained by multiple predicates; it might be related to other variables via binary (and higher-arity) predicates. How then do we arrange for an enumerative characterization of these complex constraints? How do we simultaneously solve multiple arbitrary constraints? This is clearly undecidable and we can have no such general solver. But this does not deter us; after all we are already in the business of "interactively proving" theorems about recursive definitions, an undecidable endeavor. Taking this as our definite goal, continuing further on the path we have already taken, we will give the user an even more central and explicit role in the activity of "interactively disproving" non-theorems.

What form will the user input take to (enumeratively) characterize arbitrary constraints? In case of monadic type constraints, we employed enumerator functions. These do not suffice in the presence of arbitrary-arity predicates; it is not clear what form the enumerator will take. The existence of multiple predicates constraining a single variable also causes problems: how do we combine the enumerators of the multiple predicates to obtain an enumerator that simultaneously satisfies them? The form of enumerator functions is too unwieldy to point to elegant answers to these questions.

Towards this objective, we have found useful the notion of a *fixer* which we motivate with an example in Section 6. A fixer function associated with a predicate usually takes the same arguments as the predicate,

and instead of returning a boolean true or false, *fixes* one or more arguments and returns them, such that the updated arguments satisfy the predicate. The user both provides such a function for a given predicate and further characterizes a fixer as preserving certain other relations/predicates, *e.g.*, `remove-duplicates` acts as a fixer for the `no-duplicatesp` predicate and it preserves the predicate `orderedp`, the property that a list is ordered. We sketch an algorithm that takes this information into account and orchestrates the fixers to simultaneously solve multiple predicates/constraints. Thus, the notion of a fixer wields an elegant solution to the problem of compositionally combining solutions of multiple arbitrary predicates. This has the potential to vastly improve the efficacy of property-based testing of invariants and conjectures with complex pre-conditions.

We have implemented our method in ACL2s, the ACL2 Sedan [8], which uses ACL2 [19] as the core reasoning engine. It is of principal importance that the user interface enabling "non-theorem disproving" be as intuitive and unobtrusive as possible. The user access to Cgen, the framework/tool implementing our method, is indeed so; in ACL2s, we have arranged the counterexample generation support to be automatic and invisible. We have successfully used ACL2s, with Cgen playing a crucial pedagogical role, in a classroom setting for many years now; this has served as an important evaluation for Cgen, resulting in continuous improvements in usability and engineering.

We also compare our implementation in ACL2s with Alloy [1], a popular specification analysis tool. We modeled various Alloy examples in ACL2s and analyzed them with Cgen.

It is worth pointing out that interactive theorem provers differ from theorem provers that provide decision procedures for (decidable) fragments of logic. A notable class of such theorem provers is the class of Satisfiability Modulo Theories (SMT) solvers. Notice that in the case of such decision procedures, there is a simple way to go from proving to disproving because for decidable fragments of logic, a decision procedure for satisfiability (or falsifiability) yields a decision procedure for validity and conversely. This is not the case with interactive theorem provers because they are concerned with undecidable fragments of logic and the methods used to construct a proof cannot directly be used to falsify conjectures.

Proposal Summary For my dissertation, I propose to investigate interactive non-theorem disproving. I argue that this research is important—users spend a significant amount of time debugging flawed models, abstractions, invariants and conjectures. The potential to improve the user experience of ITPs, to more completely assist all aspects of mathematical reasoning using disproving, is vast. Towards this objective, I claim that a simple and intuitive method based on concrete construction and executability, with user-specified enumerative characterizations and employing property-based testing is promising. Implementation and experiments in ACL2 Sedan and extensive use of the method in the classroom to teach formal reasoning provide support to my claims.

Proposal Structure The rest of this proposal is organized as follows. In Section 2 we describe completed work on counterexample generation, implemented as the Cgen tool. The primary interface to "program" Cgen is provided by Defdata, a language and framework for specifying data definitions; Defdata is described in Section 3. We evaluate Cgen in section 4 and present related work in section 5. In section 6, we highlight the current limitations of Cgen and motivate the notion of fixers to solve/enumerate richer constraints. Future proposed tasks and a tentative schedule are given in Section 7.

2 Counterexample Generation in ACL2 Sedan

In this section we introduce Cgen, the tool that implements counterexample generation support, *i.e.*, non-theorem disproving, in ACL2. Cgen synergistically combines property-based testing (a la Quickcheck) and the full power of the ACL2 theorem prover engine. With the help of examples, we will first describe how Cgen is used, in particular how the user "programs" Cgen and finally we describe how Cgen works, in both its default and incremental search modes.

2.1 Cgen – User Interface

To illustrate the use of Cgen[3], let us consider the triangle problem, a classic example used in software testing literature. The user wishes to formalize triangles and reason about them. The first step is to define what a triangle is. The mathematical definition of a triangle is as follows; any triple of positive integers can serve as the side lengths of an integer triangle as long as it satisfies the triangle inequality. A reasonable model is to view it as a list of three positive integers (recognized by type predicate `posp`), representing its three sides, with each side less than the sum of the other two sides. The user defines the structural aspect (*i.e.*, triple) of a triangle using the *defdata* facility and the triangle inequality using a regular predicate function. Defdata is described in more detail in the next section, it suffices for now to note that it automatically generates the predicate and enumerator definitions, *e.g.*, `triplep` and `nth-triple` in this case.

```
(defdata triple (list pos pos pos))

(defun trianglep (v)
  (and (triplep v)
       (< (third v)  (+ (first v)  (second v)))
       (< (first v)  (+ (second v) (third v)))
       (< (second v) (+ (first v)  (third v)))))
```

The user has specified a predicate describing how a triangle is to be recognized; this much he has to do, if he wants the theorem prover to assist him in reasoning about triangles. To equip Cgen to assist him, he can additionally provide an enumerator, that specifies how triangles are to be constructed.

```
(defun triangle-enum (n)
  (b* (((list p n) (split 2 n))
       ((when (< p 5)) (list 1 1 1))
       ((list n1 n2) (split 2 n))
       (lo 1)
       (hi (+ p (- 2)))
       (x1 (nth-integer-between n1 lo hi))
       (hi (+ p (- x1) (- 1)))
       (x2 (nth-integer-between n2 lo hi))
       (x3 (+ p (- x1) (- x2))))
    (list x1 x2 x3)))

(register-type triangle
               :predicate trianglep
               :enumerator triangle-enum)
```

Notice how much more involved it is to describe a function (in a pure state-less language) that explicitly enumerates triangles. Not only is it involved, a fair amount of creativity occurs in the solution preceded by some insight into the nature and form of the objects under study. We suppose, after a little bit of thought and experimentation, the user arrives at the following strategy to lay out (almost) all possible triangles: choose a perimeter, and then choose a partition of the perimeter to get the three sides of the triangle. How to choose? Our input is a number n and the purpose of the function to output the nth triangle. All our choices therefore arise from this input number n. Using the library function `split`, we bijectively split the number n into a pair of numbers (p, n). Observe that the only triangle of perimeter less than 5 is the equilateral triangle of size 1. After taking this into account, we further split n, bijectively, into two more numbers; this

[3]Cgen is enabled by default in ACL2 Sedan. In a vanilla ACL2 session, the following form enables Cgen: `(include-book "acl2s/cgen/top" :dir :system)`

lets us choose a 3-way partition of p, using the library function `nth-integer-between`. Some care is taken to make sure that each partition/side is at least of length 1.

We use `register-type` (available from Defdata) to inform Cgen that `triangle-enum` is a function that can be used to enumerate objects satisfying predicate `trianglep`.

While not a bijection, we would like for `triangle-enum` to be a surjection, *i.e.*, it should enumerate the set of all integer triangles. This fact is of course very difficult to prove and we trust the user to use his model-specific insight to provide such a function.

A less painful alternative is for the user to simply give a decent approximation of the enumerator for triangles, say by reusing the enumerator for triples (which was automatically generated by Defdata), and to cede the responsibility for searching for triangles of the right shape to Cgen and the theorem prover.

```
(register-type triangle
                 :predicate trianglep
                 :enumerator nth-triple)
```

This is how the user programs Cgen's search for test data objects of the correct shape that satisfy the type-like monadic predicates in the hypotheses of conjectures under test.

The actual working of Cgen is invisible to the end-user. It is automatically invoked every time the user attempts to prove a conjecture using `thm` or `defthm`, standard interfaces to ACL2. Let us suppose the user plays around with triangles of various shapes (equilateral, isosceles, scalene) and conjectures that there are no isosceles triangles whose third side is the product of the other two sides. To highlight the work done by the theorem prover, let us add the constraint that the largest side is greater than 256; this precludes the success of bounded exhaustive testing of small triangles.

```
(defun shape (v)
  (if (trianglep v)
      (cond ((equal (first v) (second v))
             (if (equal (second v) (third v))
                 "equilateral"
               "isosceles"))
            ((equal (second v) (third v)) "isosceles")
            ((equal (first v) (third v)) "isosceles")
            (t "scalene"))
    "error"))

(thm
 (implies (and (trianglep x)
               (> (third x) 256)
               (= (third x) (* (second x) (first x))))
          (not (equal "isosceles" (shape x)))))
```

Here is the output snippet due to Cgen:

```
We tested 2000 examples across 2 subgoals, of which 897 (897 unique)
satisfied the hypotheses, and found 897 counterexamples and 0 witnesses.

We falsified the conjecture. Here are counterexamples:
 [found in : "Subgoal 3'''"]

 -- ((X '(259 1 259)))
 -- ((X '(327 1 327)))
 -- ((X '(264 1 264)))
```

In the above example, we used `triangle-enum` as the enumerator for triangles. The use of a user-provided enumerator was not essential, as the above `thm` successfully finds counterexamples even with the naive `nth-triple` enumerator for triangles.

This is how Cgen is used, tightly integrated with the existing interfaces of ACL2 to prove conjectures. The summary output generated by Cgen is in English and self-explanatory. There is also a programmatic interface to Cgen, for people who want to build tools on top of Cgen. The documentation for Cgen and its ecosystem and can be found online or within an ACL2 session using the command `:doc cgen`.

2.2 Cgen – Integrating property-based testing

In this section, we provide an overview of how Cgen works, its design and implementation. Cgen operates in two search modes. We will first describe the default mode, and then explain the incremental search strategy.

Cgen integrates property-based testing and the proof-based search of the theorem prover. To get a better appreciation, let us look at what happens when we do not take the help of the theorem prover, *i.e.*, when we only use property-based testing. Instead of using `thm`, we will invoke a user interface specific to Cgen, `test?`, that in its naive setting, performs stand-alone property-based testing, with no interaction with the theorem prover.

```
(acl2s-defaults :set testing-enabled :naive)

(test?
 (implies (and (trianglep x)
               (> (third x) 256)
               (= (third x) (* (second x) (first x))))
          (not (equal "isosceles" (shape x)))))

 ==>

**Summary of Cgen/testing**
We tested 1000 examples across 1 subgoals, of which 0 satisfied
the hypotheses, and found 0 counterexamples and 0 witnesses.

Test? succeeded. No counterexamples were found.
```

As you can see, on its own, property-based testing is unable to find any counterexamples to the same conjecture. In its naive setting, Cgen fails miserably, because even though it picks up the right type of x and only instantiates integer triangles, it is very hard to satisfy the extra constraints on x. The probability that a randomly sampled triangle turns out to be isosceles, with its longest side greater than 256 and equal to the product of the other two, is indeed very low, even with the favorable assumption that the sample space of triangle side lengths is uniformly distributed between 1 and 500.

In its default search mode, Cgen uses the full power of the ACL2 theorem prover to simplify conjectures for better testing. The main idea is to let ACL2 use all of the proof techniques and the database of lemmas (rules) at its disposal to simplify conjectures into subgoals, and then to test those subgoals. For the above example, the prover opens up the definitions of `shape` and `trianglep` and uses case analysis, reducing the above top "Goal" to three subgoals. After several simplification steps and a few rounds of destructor elimination, `Subgoal 3'''` easily yields a counterexample on application of property-based testing.

```
We tested 3000 examples across 3 subgoals, of which 725
satisfied the hypotheses, and found 725 counterexamples.
```

```
We falsified the conjecture. Here are counterexamples:
 [found in : "Subgoal 3''''"]
(IMPLIES (AND (INTEGERP X1)
              (< 0 X1)
              (< 1 (* 2 X1))
              (< 256 X1))
         (EQUAL X1 1))

 -- ((X '(354 1 354)))
 -- ((X '(320 1 320)))
 -- ((X '(263 1 263)))
```

Notice that the theorem prover simplified away two variables representing two sides of the triangle, thus drastically simplifying the constraints; the probability of finding a counterexample went up from very low to 1 (for the above subgoal, Cgen infers that x is an integer greater than 256). Apart from case-analysis and destructor elimination, the primary simplification is due to the presence of libraries of lemmas (notably arithmetic-5 [24]). In this respect interactive theorem proving has a huge advantage over other tools routinely combined with testing, especially considering the fact that most interactive theorem provers have good library support.

This gives a flavor of the high-level working of Cgen. We now give an overview of its design and implementation.

Cgen can be use with any ITP that satisfies a set of weak assumptions. One of these assumptions is that the ITP consists of a recursive loop, where each step corresponds to the application of a proof technique that simplifies the current goal P into simpler subgoals. Property-based testing can be applied before each such prover step using the procedure `cgen-search`. In our implementation, instead of testing each subgoal, we only apply `cgen-search` to certain subgoals, called "checkpoints" in ACL2, that are important milestones in the ACL2's proof search. The actual integration is implemented in ACL2 using override-hints [20], with `cgen-search` registered as a callback procedure. Relevant metadata, intermediate results and statistics are accumulated in `context`. A user-defined stopping condition specifies when the loop (implemented as procedure `prover-loop/cgen`) is to be aborted. The result is one of three conditions: *valid*, *falsified*, *dont-know*.

```
def prover-loop/cgen (P context)
  context = update ITP type info for vars of P
  context = (cgen-search hyps(P) concl(P) context)
  if stopping-condition(context)
     abort/return result(context), context
  Subgoals, context = (prover-step P)
  for each s in Subgoals
      (prover-loop/cgen s context)
  return result(context), context
```

Simple Search

The `cgen-search` procedure, in the simple search mode (the default mode of operation), is comprised of three main tasks; first compute an enumerable/parameterized sample space for each variable of the property under test, then instantiate the variables with randomly sampled[4] values to obtain an assignment and finally

[4]Random sampling is the default, with a pseudo-geometric distribution used for infinite spaces and a uniform distribution for finite; the alternative option is bounded-exhaustive sampling.

execute/evaluate the conjecture/property under the concrete assignment. The hypotheses and conclusion are tested separately to record both counterexamples and witnesses; this also helps filter out instances that are vacuously true, *i.e.*, when the hypotheses evaluate to false.

The first task of `cgen-search` is the most crucial: infer, to the best of its capability, all the *enumerable constraints* and arrange their enumerative descriptions. By enumerable constraints we mean those constraints that can be solved completely, *i.e.*, whose solutions are expressible using enumerative descriptions. At present, this capability is limited to monadic type-like constraints, integer and rational ranges, variable equality and variable membership constraints. This capability is programmable; the primary user interface to program Cgen is provided by the defdata facility that is described in the next section. In section 6, we show how the notion of a fixer enables a user to extend the capability to richer constraints. An enumerative description takes one of three possible forms: internal Cgen encoding, defdata expression and ACL2 expression.

```
def cgen-search/simple (hyps, concl, context)
    type-alist = compute-strongest-type-info(hyps, concl, context)
    enum-desc-alist = arrange-enum-desc(hyps, concl, type-alist, context)
    symbolic-σ = parameterized-assignment(enum-desc-alist, context)
    N = num-trials(context)
    repeat N times
      concrete-σ = instantiate enumerating params in symbolic-σ
      evaluate hyps and concl under concrete-σ
      record/classify result in context
    return context
```

The actual task of computing enumerable descriptions is performed by `arrange-enum-desc`, taking into account, the form of the constraints in the hypotheses, the strongest type information available from the ITP proof context and the correspondence between constraints and its enumerable description previously recorded in Cgen. The latter information is given by the user, who programs Cgen to enumerate/solve certain constraints (usually via Defdata). How the resulting enumerable descriptions are arranged is explained later in the section (Select procedure).

Once each variable of the property is associated with an enumerative description, `parameterized-assignment` performs a straightforward syntax-directed translation to build the corresponding ACL2 value expression dependent only on random parameters that are used to enumerate the range of actual solution space. The resulting symbolic assignment, *i.e.*, a symbolic value binding, is then repeatedly instantiated by (randomly) varying the enumerating parameters and the resulting concrete assignment is used to evaluate the hypotheses and conclusion separately. The result of each such test is recorded in the context.

Incremental Search

Consider the following constraints on integers, $x1 > x2, x1 < x2 * x2$. These inequalities impose a binary relationship between $x1$ and $x2$. But this relation is not undirected in the sense that there is an asymmetry and direction introduced by the functional term in the second inequality. This functional term by means of its computational cost, introduces a dependency between x1 and x2. It is computationally easier, to solve for the value of x1 given a feasible partial solution for x2, than the other way round. This motivates an ordering among variables determined by some notion of computation cost. In the above example, $x2$ ought to be selected and assigned first. Suppose $x2$ is assigned the value 3, the resulting constraints are monadic range constraints on $x1$: $x1 > 3, x1 < 9$. Ranges, *i.e.*, inequality constraints that involve a single variable and constants (hi, lo) are obviously enumerable and natively supported by Cgen. Thus what was before outside Cgen's scope, is, after assignment and simplification (concrete evaluation), natively understood.

This motivates the incremental search mode of Cgen; instead of assigning symbolic values to all variables en block, and then repeatedly instantiating and testing it, one proceeds incrementally. We *select* the most independent, least constrained, variable first, *assign* it a concrete value (as before), *propagate* this partial assignment to simplify the constraints using the full power of the ITP and iterate till a complete assignment is found, at which point, the property under test is evaluated. If propagation results in inconsistency, *i.e.*, one of the hypotheses is simplified to `nil`, then we backtrack. The algorithm in more detail is presented in [9]; below we describe the `select` procedure.

The `select` procedure operates on P, the property under test. If a variable is constrained to be equal to a constant, it is returned. Then `select` uses congruence closure to handle variable-variable equalities (*e.g.*, $x_1 = x_2$) in P. Functional dependency is modelled using a directed graph. The graph is topologically sorted and the least dependent variable is returned; a similar analysis is used to determine how enumerable descriptions are arranged in the right order by `arrange-enum-desc`.

```
def select (P)
    if x = c in hyps(P) // c is a constant
      return x
    do congruence closure on P
    G = build-dependency-graph(P)
    X = topological-sort(G)
    return last(X)  // most independent
```

The dependency graph, whose nodes are the free variables of P, is built as follows: for each constraint φ of P, we apply the following rules, based on its form; if more than one rule applies, the earliest rule is used. In the rules, we assume that x and y are (distinct) free variables of P and *term* is inductively defined to be either a variable, a constant expression, or a function application with arguments that are *terms*. Terms that are function applications are denoted by *fterm*.

1. If φ of the form $x \bowtie$ *fterm* such that $\bowtie$ is a binary relation, $y \in \mathsf{freeVars}(\mathit{fterm})$ and $x \notin \mathsf{freeVars}(\mathit{fterm})$, we add an edge from node x to node y, recording that x "depends" on y. *Intuition*: It easier to satisfy $x \leq 2^5$, simply evaluate 2^5 and solve $x \leq 32$. Satisfying $32 \leq 2^y$ is trickier.

2. If φ is of the form $x \in y$, then add an edge from x to y.

3. If φ is of the form $x \bowtie y$ where $\bowtie \in \{<, \leq, >, \geq\}$ we don't add an edge. *Intuition*: Constraint $x > 3$ is as easy to satisfy as $42 > y$, so we avoid complicating the graph. (This overrides the following rule.)

4. If φ is of the form $R(\mathit{term}_1, \mathit{term}_2, \ldots, \mathit{term}_n)$, such that $x \in \mathsf{freeVars}(\mathit{term}_i)$, $y \in \mathsf{freeVars}(\mathit{term}_j)$, $i \neq j$, $n \geq 2$ and R is an arbitrary n-ary relation, then we simply add a bidirectional edge, recording their mutual dependence, but giving no preference (locally) to either variable.

We have described the Cgen utility, its user interface and its design. We will now highlight Defdata, an important constituent of the Cgen framework.

3 Defdata – Enumerating data definitions in ACL2

Data definitions are an essential part of designing programs and modeling systems; they form the first and the most basic means of describing the structure of the objects under study. Only after the structural aspects of data are defined, does the user refine it with additional constraints. Thus it is imperative to have a handle on data definitions, both with respect to the theorem proving and non-theorem disproving.

In ACL2, data definitions are described using a predicate function, which provides an analytic characterization of the data, *i.e.*, the input element is analyzed if it matches the desired structure, deconstructed and

the constituent elements are (recursively) analyzed. This much the user has to do to convey the structure of the data to the theorem prover. Since our primary concern is to do with non-theorem disproving using concrete data and calculation, we need to do more; in particular we need to describe the same data bottom-up, in a constructive manner, and obtain an enumerative characterization.

The defdata framework provides a comprehensive solution: by introducing a high-level, intuitive language to describe data, we are able to obtain both the views, the predicative and the enumerative. The `defdata` macro provides the primary user interface to "program" Cgen with the capability of enumerating solutions to type-like monadic constraints; moreover, the defdata language expressions serve as enumerative descriptions and are used as such in the implementation of Cgen.

Here is a quick demonstration of the defdata language. Suppose the user is modeling a filesystem, the key concepts can be defined in the defdata language as follows.

```
(defdata inode nat)

(defdata file (cons inode string))
(defdata files (listof file))

(defdata
  (dir (oneof nil (cons (cons string dir-entry) dir)))
  (dir-entry (oneof file dir)))
```

When submitted, the defdata event introduces a new typename, predicate and enumerator definitions for the type, and a host of other events that support a "typed" language setup. For example, given the last form, ACL2s will automatically generate *type predicate* `dirp`, which recognizes a directory. It supports Cgen (property-based testing) by generating `nth-dir`, a type enumerator that maps natural numbers into directories[5]. In general, a *type enumerator* for type T is a surjective function from natural numbers to T. ACL2s will automatically generate a type enumerator for any new data types defined using the data definition framework. The type predicate and enumerator are generated using a syntax-directed translation.

The defdata framework provides enumerators for the primitive types and for basic ACL2 types that are combinations of primitive types (*e.g.*, the natural numbers, integers, rationals, and lists). Each data object in the ACL2 universe is treated as a singleton "type", *i.e.*, a set with just one element, the data object itself. The type which represents all of the ACL2 universe is called *all*; every type is thus a subset of *all*. Also fully supported are user-defined union types, product types, list types, alist and map types, range types, record types, and enum (member) types. In particular, Defdata framework supports custom types, *i.e.*, arbitrary monadic predicates (*e.g.*, prime numbers), with the burden of generating the enumerator resting on the user.

Defdata performs an additional crucial service for Cgen: it helps encode precise type information and type relations in ACL2's type-reasoning component[6]. It does so by generating theorems that characterize the type relations such as inclusion (subtype) and exclusion (disjoint); and it generates events that support polymorphic type reasoning.

Given a data definition `(defdata M` s`)`, defdata computes the set of rules that completely characterize the inclusion/exclusion type relationship between M and typenames in s. Defdata thus automatically updates the subtype type lattice for every new data type. This is very useful, not only does this enable Cgen to query the lattice and infer minimal type information, it also enables automated type-like reasoning.

Polymorphic support in Defdata adds significant user convenience. Indirectly, it also helps Cgen infer more precise type information. Consider the list concatenation function `append`, if the user specifies a polymorphic type signature for it (see below), then Defdata sets up the appropriate theory (using encapsulation

[5]Apart from these events, the predicate and enumerator for dir-entry is also generated, and various other type relationship lemmas.

[6]The *Tau system* is a component of the ACL2 theorem prover, designed to quickly prove "type-like" proof obligations. It is driven by a database derived from previously admitted lemmas and definitions.

and instantiated tau-rules) such that, the append of two rational lists is determined to be a rational list, instead of just a (generic) list. This allows Cgen to infer a stronger type for such functional terms and consequently results in a more efficient property-based testing.

Here are some example polymorphic signatures using Defdata's `sig` macro.

```
(sig append ((listof :a) (listof :a)) => (listof :a))

(sig nth (nat (listof :a)) => :a
     :satisfies (< x1 (len x2)))

(sig put-assoc-equal (:a :b (alistof :a :b)) => (alistof :a :b))
```

A Defdata tutorial and a more complete description of the framework is available elsewhere [7].

4 Evaluation

First, we present our experience in using ACL2s in the classroom setting which has served as an important evaluation for Cgen and has led to continuous improvements in it's usability. In the next subsection, we evaluate Cgen by comparing it to Alloy. This comparison is described in more detail in the Cgen paper [9]. We modeled various Alloy examples in ACL2s and find counterexamples to all failed properties (falsified by Alloy) using Cgen.

4.1 Classroom experience

For several years, we have been teaching freshman students at Northeastern University how to reason about programs. We have used ACL2s and it has been an invaluable teaching aid. One place where students often struggle is in writing specifications. They sometimes make logical and conceptual mistakes, and they often omit required hypotheses. Therefore, they often try to prove conjectures that are false. In large part, the motivation for this work was to help students by providing them with counterexamples. In this, we have succeeded because the Cgen framework tends to find counterexamples easily. The counterexamples allow students to see what is wrong with their conjectures, in terms they readily understand. Without the counterexamples, students are left trying to determine whether they need more lemmas or whether their conjectures are false, a skill that takes time to develop.

4.2 Comparison with Alloy

Alloy [17] is a declarative modeling language based on sets and relations, primarily used for describing high-level specifications and designs. Alloy Analyzer [18] is a tool that supports automatic analysis of models written in Alloy. Given a bound on the number of model elements, called *scope*, the Alloy Analyzer (AA) translates Alloy models (and its specifications) into Boolean formulas, uses state-of-the-art SAT technology to generate satisfying instances and translates them back to corresponding set and relation instances of the objects in the model. Alloy is based on a first-order relational logic with transitive closure, which allows expressing rich structural properties using succinct expressions.

We analyzed 10 properties from 3 Alloy problems (specifications), all from the Alloy book [17] and can alternatively be downloaded from the Alloy distribution. Table 1 shows results, comparing the performance of our method implemented in ACL2s, with the performance of the Alloy Analyzer (AA). The time (in seconds) is measured on an Intel Core i3, 2.8GHz, 4GB memory machine. The Alloy analysis time is the total of the time spent on generating CNF and solving it using the SAT4J solver. The time taken by our method is what the ACL2 macro `time$` reports and includes the time taken by the ACL2 theorem prover. The Scope column for AA either denotes the minimum scope that finds a counterexample, or the maximum

Property	Alloy Analyzer			ACL2s/Cgen	
	Scope	Time	Result	Time	Result
delUndoesAdd	25	26.41	–	0.07	QED
addIdempotent	25	37.76	–	0.19	QED
addLocal	3	0.08	CE	1.35	CE
lookupYields	3	0.05	CE	0.83	CE
writeRead	34	99.69	–	0.02	QED
writeIdempotent	33	44.13	–	0.01	QED
hidePreservesInv	61	24.91	–	0.26	QED
cutPaste	3	0.20	CE	0.49	CE
pasteCut	3	0.20	CE	1.38	CE
pasteAffectsHidden	27	117.63	–	0.42	QED

Table 1: Comparison with Alloy Analyzer (AA)

scope for which AA can check the property before exceeding the 2 minute time limit, or the 1 GB memory limit. Note that the user does not specify scope when using ACL2s; the size of the test data generated by Cgen is bounded by the pseudo-random sampling distribution which is fixed. The Result column shows either "CE", "QED" or "–", that stand for "Counterexample found", "Proof found", "Neither Counterexample nor Proof found", respectively.

The first 4 properties are from the model of an email client's address book supporting aliases and groups, the *writeRead* and *writeIdempotent* properties are from the abstract memory problem, the next 4 properties are from an Alloy model describing the design of a media file management software.

We took the above examples and modeled them in the ACL2 language, mimicking the original formulation in Alloy as much as possible. In particular, we used *set* types and *map* types (*i.e.*, binary relations), which are part of defdata language. These respectively make use of the ordered sets library and the records library [27, 21, 12] in the ACL2 standard library distribution. These libraries provide a generic collection of reasoning rules (used in rewriting) about sets and records. In fact they are powerful enough to prove all the properties that Alloy exhaustively checked within the scope.

5 Related Work

We classify the related work in disproving conjectures by the techniques used and in the end summarize previous efforts to support counterexample generation in ACL2. We restrict our attention to work targeting rich specification languages that do not fall under a decidable fragment of logic.

Property-based random testing

Executable specifications (properties) can be randomly instantiated (values generated for its free variables) and tested. This is known as property-based random testing and the success of QuickCheck [11] has led its wide-spread adoption in functional programming languages. The ITP community has also embraced property-based testing, for example in Isabelle [2, 5], Agda [15], PVS [30] and more recently in Coq [14].

Usually, Quickcheck-like tools provides a library of combinators for users to write testdata generators by hand, although some support automation for type-like pre-conditions exists [5, 23]. The SciFe [25] framework in Scala, provides higher-order combinators to mechanize construction of efficient enumerators for even complex data structures with invariants.

Reduction to SAT/SMT

The other standard technique for generating counterexamples for a conjecture is to use a SAT or SMT solver. This requires translating from a rich, expressive logic to a restricted decidable logic.

The commonly used MACE-style [28] translation bounds the cardinality of the atomic types and encodes predicates and functions using propositional variables. Refute [36] and Nitpick [4] are SAT-based disprovers available in Isabelle, that use sophisticated encodings of higher-order logic.

The Leon verification system [3], uses a pure functional subset of Scala as its specification language. Its core solver builds on the SMT-solver Z3 [13] and adds support for recursive functions using an iterative loop, that incrementally unfolds the function and alternates between an under-approximation (non-recursive control branches) and an over-approximation (using uninterpreted functions). More recent work on handling recursive functions presents a translation that instead of iteratively unfolding and calling a SMT-solver, uses quantifier instantiation and invokes the solver only once [31].

Symbolic-based approaches

Other approaches to disproving include symbolic techniques based on narrowing or logic programming. Narrowing-based testing consists of symbolic evaluation of the conjecture with partially instantiated terms and then to incrementally refine (*i.e.*, instantiate) these when needed. Prominent tools implementing this include LazySmallCheck and Isabelle's Quickcheck [26, 32, 5]. The latter also provides "smart generators" to handle conditional conjectures, *i.e.*, properties with preconditions, where the test data generator associated with a given precondition is synthesized from the mode-analysis performed on the set of Horn clauses obtained from it.

ACL2

Generating counterexamples to give better feedback in ACL2 and its predecessors has motivated many efforts, starting with Won Kim's thesis [22], and more recently: DoubleCheck [16], Pythia [33] and SAT Checking [34]. Kim's system used heuristic-based constraint-solving techniques in conjunction with limited simplification (rewriting and definition expansion) to construct examples (assignments to free variables in the constrained formula). Pythia uses the SAT-based tool Alloy Analyzer [18] to generate small counterexamples for simple ACL2 formulas. While Pythia works for simple formulas, it is impractical for formulas which contain references to recursive definitions; in such cases, the user needs to write Alloy models. Sumners [34] uses SAT checking to generate counterexamples and faces the similar problem of being amenable only to very simple formulas. DoubleCheck is a QuickCheck-like library implemented in DrACuLa [35], a DrScheme-based GUI for ACL2. Since it is not written in ACL2, and does not have an integrated data-definition framework, it places the burden of specifying data generators and random-data-distributions on users.

6 Fixers – Solving arbitrary primitive recursive predicates in ACL2

Consider the following scenario, which is outside the scope of the current Cgen implementation, *i.e.*, even after full simplification by the ITP, the resulting constraints are not in a form enumerable by Cgen.

We want to mechanically prove the correctness of Dijkstra's shortest path algorithm (DSP). The specification of the algorithm is this: Given a graph G, and two nodes, a and b, the path (if it exists) computed by DSP is the shortest path from a to b. The DSP algorithm is a greedy algorithm that traverses the graph from the source node a maintaining a table of shortest paths from the source to each visited node. After visiting all reachable nodes, it simply looks up in the table the path stored for b. The proof as given in textbooks involves an invariant on the path table that it contains the shortest paths (from a) in the subgraph obtained by restricting the input graph to the visited nodes. Although the crux of the proof is to come up with the main invariant, it is a fairly non-trivial exercise in interactive theorem proving [29, 10]; it requires the user to discover and prove numerous supporting lemmas (*e.g.*, 57 domain-specific lemmas in the ACL2 proof). Consider the following lemma in the ACL2 proof:

```
(implies (and (graphp g)
              (path-tablep a pt g)
              (path u pt)
              (member v (neighbors u g)))
         (pathp-from-to (append (path u pt) (list v))
                        a v g))
```

It says that, for a graph `g`, a path-table `pt` contains paths from node `a` to vertices in `g`, if there is a path (from `a`) to node `u` in `pt`, and node `v` is a neighbor of `u`, then extending that path by `v` is a path from `a` to `v` in `g`. This is a theorem and ACL2 automatically proves it by induction if it has access to a handful of useful lemmas (rewrite rules).

If the user has a fairly good idea of the model and its behavior, designs the right concepts (functions), formulates the right invariants, provides the right lemmas and hints to limit the search space, then it is reasonable to expect the prover to find a complete formal proof successfully. We are primarily concerned with how to increase the usability of the prover when these conditions do not hold, which in our experience is almost always. Until the proof is completed, the user might not be entirely familiar with the problem, might be considering and designing supporting concepts, might make mistakes in the formalization, devise invariants that are not valid, or might just make silly human errors.

Suppose, for example, the user makes the following mistake in the above lemma:

```
(implies (and (graphp g)
              (path-tablep a pt g)
              (path u pt)
              (memp v (neighbors u g)))
         (pathp-from-to (append (path u pt) (list u v))
                        a v g))
```

Perhaps the user forgot how he has encoded paths in the path table. Mistakenly thinking the path from `a` to `u` stored in the path-table is not inclusive (*i.e.*, omits `u`), he adds `u` before its neighbor `v`. A sloppy mistake, but a perfectly reasonable one to manifest in the initial process of formalization and design. After about 50 seconds and 80 subgoals, the prover returns with a long failed proof output, printing out 5 useful subgoals (checkpoints) to look at.

Cgen, employing type-based enumerators, also fails to generate a counterexample. Let us analyze the reasons for the failure. Note that at the top-level goal, the only enumerable constraints are the membership constraint and any defdata type constraints that Cgen can glean from the ITP context. Even if Cgen perfectly infers the monadic type constraints on all the variables in the conjecture, that `pt` is a path-table (a map from vertices to list of vertices), `a,u,v` are vertices, `g` is a adjacency-list, mapping vertices to weighted edges, it is plain to see why property-based testing from these constraints alone fails; the variables in the conjecture are highly interdependent on each other and the monadic constraints do not take this dependency into account. For constraint `(path-tablep a pt g)`, the probability that a randomly generated path-table `pt` will only contain paths originating from `a` in a randomly generated graph `g` is close to 0. For `(path u pt)`, the probability that a generated vertex `u` will have a path from `a` in `g` and that the path is stored in `pt` is also close to 0. Moreover, that the generated assignments will simultaneously satisfy all the constraints is still less probable.

It is left to the user, to stare at the conjecture, stare at all the predicate definitions, stare at the checkpoints in the failed proof and break the stalemate. There is a huge difference between knowing and not knowing whether a conjecture is false. In the former, one can put on their meanest counterexample generator hat and conjure up a series of examples that satisfy the hypotheses, but not the conclusion and then manually test them for validation. How can we can get the ITP to help us put on this hat?

The fact of the matter is that `g, pt, a, u` and `v` are intricately tied together, and the only way to generate their assignments is to generate them together, *i.e.*, to solve the above constraints simultaneously at their origin. But these are not linear equations that we might find ready techniques of simultaneous equation solving. Each predicate in the constraints above is a concept that is recursively defined and quite non-linear we might say. To solve them is to have a solver for arbitrary (primitive) recursive constraints. This is clearly undecidable and we can have no such general solver. So we ask, how might we "interactively solve" simultaneous, arbitrary, primitive recursive constraints?

To this objective we have found useful the notion of a *fixer*, that is more flexible and general than that of an enumerator. It is cumbersome to generalize enumerators to binary and higher-arity predicate constraints, moreover, it is not clear how one can combine enumerators to solve simultaneous constraints. Fixers provide an elegant solution to the problem of simultaneously solving multiple, higher-arity constraints. A fixer is a function associated with a predicate, that instead of saying "yes" or "no", provably computes the "yes" instances, usually by minimally fixing one (or more) errant argument(s). Furthermore, applying a fixer to certain other predicate constraints, leaves them invariant, *i.e.*, a fixer preserves the truth of a collection of predicates. The idea of the solution is that the user informs Cgen of these relations, and Cgen orchestrates the appropriate enumerators and fixers to simultaneously solve/enumerate the constraints under test.

Let us demonstrate how a user would use fixers to interactively disprove the above flawed conjecture. Let us first see how we can individually solve the aforementioned problematic constraints. To solve `(path-tablep a pt g)`, we will look at its predicate definition and then design a fixer function for it.

```
(defun path-table-p (a pt g)
  (if (endp pt)
      t
    (b* (((cons v path) (first pt)))
      (and (or (null path)
               (pathp-from-to path a v g))
           (path-table-p a (rest pt) g)))))
```

`path-table-p` holds when for every entry `(v,path)` in the table `pt`, if path is not empty, then it is a path in `g` from `a` to `v`. Below we define a fixer function to piecewise reconstruct/fix `pt` to satisfy the predicate. Notice that it re-uses the recursive structure of the predicate. We assume the user has specified a fixer for `pathp-from-to` predicate.

```
(defun path-table-fix (a pt g)
  (if (endp pt)
      '()
    (b* (((cons v path) (first pt)))
      (cons (cons v (if (< (len path) 2)
                        '()
                      (pathp-from-to-fix path a v g)))
            (path-table-fix a (rest pt) g)))))
```

Now, the user, under some preconditions, has to prove that `path-table-fix` is indeed a fixer for `path-table-p`. The appropriate rule/lemma would be:

```
  (alistp pt) & (graphp g) & (member a (strip-cars g))
   =>
  (path-tablep a (path-table-fix a pt g) g)
```

This rule informs Cgen how to satisfy the path-table constraint using `path-table-fix`. The preconditions specify that, `pt` is an association list (*i.e.*, a table), `g` be a graph, and `a` is in the vertices of `g`. Note that all

the preconditions are enumerable; Cgen can satisfy (solve) all of these and then apply the fixer function to satisfy `path-table-p`.

Let us now deal with the other constraint `(path u pt)`; it says that there is an entry in `pt` starting with vertex `u`. Like above, we can design a fixer for this constraint, and then specify the corresponding rule. Instead, we will proceed on more general lines. We will use the fact that membership constraints have first-class status in Cgen and directly use the following transformation rule. It says that to satisfy `(path u pt)`, we simply fix `u` to be a member of the "keys" of `pt` (In ACL2, `strip-cars` is used to find the keys of a key-value association list).

```
(path u pt) --> (member u (strip-cars pt))
```

Each such rule provided by the user, transforms a predicative characterization of a variable's search space into a computational, enumerative characterization of the search space. Thus we now have a handle, via Cgen, to solve the problematic constraints individually. The only remaining question is how solve all the constraints together, simultaneously.

We need to find an algorithm that orchestrates the application of fixers in such a way that each fixer not only satisfies the constraint it is applied to, but, also preserves all the previously satisfied constraints. Thus in addition to "fix" rules, which associate a fixer with the predicate it satisfies, the user also needs to specify "preservation" rules, which will inform Cgen, that applying a fixer to a true (satisfied) constraint, does not overturn its truth value. If it is possible, we would like to guarantee that at the end of the process of choosing the appropriate fixers and arranging the appropriate order of their application, all the constraints are simultaneously satisfied. If it is not possible, we would like an arrangement that would maximize the number of satisfied constraints.

In the above example, it is easy to find an arrangement of enumerators and fixers that simultaneously satisfies all constraints. We apply the enumerators first to satisfy the monadic type constraints inferred by Cgen. Then we apply `path-table-fix` to fix `a,pt,g` to satisfy path-table constraint; it is easy to see that `path-table-fix` preserves all the previously satisfied type constraints. We then solve for `u` (in `(path u pt)` by using the equivalent membership constraint (see the transformation rule above), without falsifying the `path-table-p` constraint (since `u` does not even appear in `(path-tablep a pt g)`). Finally Cgen solves the membership constraint for `v` and notice that this too trivially preserves the previous constraints.

Applying manually the above rules to the aforementioned buggy conjecture, ACL2s/Cgen is able to simulate the rest of the above solution (albeit partially, using its `select` procedure) to find counterexamples within 2 seconds of search.

7 Proposed Work and Schedule

7.1 Fixers

We propose to incorporate support for fixers within the Cgen framework, extending the capability of Cgen to enumerate arbitrary higher-arity constraints. We plan to develop a theory of fixers, and, based on it, design and implement an algorithm to orchestrate fixers to simultaneously satisfy a given set of constraints. We intend to design a MAX-SAT encoding and employ a SAT-Solver to provide an optimal solution with respect to maximizing the number of constraints satisfied/fixed. For experimental evaluation, we plan to pick 2-3 diverse non-trivial models with complex invariants and constraints from the ACL2 regression suite and use Cgen to simultaneously satisfy the hypotheses of the theorems. Finally, we will write up and submit a conference paper describing this work.

7.2 Cgen Improvements

The incremental search mode of Cgen, is not as fine-tuned as the default mode; the main stumbling block is that propagation of the partial assignment by the ITP's deduction engine often results in constraints whose

form is not directly amenable to Cgen; we plan to study this and improve the performance of incremental search. The documentation and user guide for programmatic use of Cgen/Defdata framework needs to be written. We plan to integrate Cgen with the standard libraries of the ACL2 distribution, so that Cgen is readily and more widely used by the ACL2 community.

7.3 Tentative Schedule

September 2015	Proposal
September 2015 - November 2015	Fixers (Section 7.1)
December 2015 - March 2016	Cgen improvements (Section 7.2); writing dissertation
April 2016	Defense

References

[1] Alloy home page. See `http://alloy.mit.edu/alloy/`.

[2] Stefan Berghofer and Tobias Nipkow. Random testing in isabelle/hol. In *SEFM*, pages 230–239. IEEE Computer Society, 2004.

[3] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An overview of the leon verification system: Verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala*, page 1. ACM, 2013.

[4] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[5] Lukas Bulwahn. The new quickcheck for isabelle. *Certified Programs and Proofs*, pages 92–108, 2012.

[6] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Integrating testing and interactive theorem proving. In David Hardin and Julien Schmaltz, editors, *ACL2*, volume 70 of *EPTCS*, pages 4–19, 2011.

[7] Harsh Raju Chamarthi, Peter C. Dillinger, and Panagiotis Manolios. Data definitions in the ACL2 sedan. In Freek Verbeek and Julien Schmaltz, editors, *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014.*, volume 152 of *EPTCS*, pages 27–48, 2014.

[8] Harsh Raju Chamarthi, Peter C. Dillinger, Panagiotis Manolios, and Daron Vroon. The ACL2 Sedan theorem proving system. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2011.

[9] Harsh Raju Chamarthi and Panagiotis Manolios. Automated specification analysis using an interactive theorem prover. In Per Bjesse and Anna Slobodová, editors, *FMCAD*, pages 46–53. FMCAD Inc., 2011.

[10] Jing-Chao Chen. Dijkstra's shortest path algorithm. *Journal of Formalized Mathematics*, 15:144–157, 2003.

[11] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[12] Jared Davis. Finite set theory based on fully ordered lists. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '04)*, November 2004.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for coq. In *The Coq Workshop*, 2014.

[15] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2003.

[16] Carl Eastlund. DoubleCheck your theorems. In *8th International ACL2 Workshop*. ACM, 2009.

[17] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[18] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In *ICSE*, pages 730–733, 2000.

[19] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[20] Matt Kaufmann and J Strother Moore. Override-hints documentation. See `http:www.cs.utexas.edu/users/moore/acl2/current/OVERRIDE-HINTS.html`.

[21] Matt Kaufmann and Rob Sumners. Efficient rewriting of operations on finite structures in ACL2. In *Third International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 '02)*, April 2002.

[22] Myung Won Kim. On automatically generating and using examples in a computational logic system, 1986. See `ftp://ftp.cs.utexas.edu/pub/boyer/diss/kim.pdf`.

[23] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In *Implementation of Functional Languages*, pages 84–100. Springer, 2003.

[24] Robert B. Krug. Arithmetic-5 library. See books/arithmetic-5/README in ACL2 regression suite.

[25] Ivan Kuraj and Viktor Kuncak. Scife: Scala framework for efficient enumeration of data structures with invariants. In *Proceedings of the Fifth Annual Scala Workshop*, pages 45–49. ACM, 2014.

[26] Fredrik Lindblad. Property directed generation of first-order test data. In *Trends in Functional Programming*, pages 105–123. Citeseer, 2007.

[27] Panagiotis Manolios and Matt Kaufmann. Adding a total order to ACL2. In Matt Kaufmann and J Strother Moore, editors, *Proceedings of the ACL2 Workshop 2002*, 2002.

[28] William McCune. A davis-putnam program and its application to finite first-order model search: Quasigroup existence problems. *Preprint, Division of MCS, Argonne National Laboratory*, 1994.

[29] J Strother Moore and Qiang Zhang. Proof pearl: Dijkstra's shortest path algorithm verified with acl2. In *Theorem Proving in Higher Order Logics*, pages 373–384. Springer, 2005.

[30] Sam Owre. Random testing in PVS. In *Workshop on Automated Formal Methods(AFM)*, volume 10, Seattle, WA, USA, 2006.

[31] Andrew Reynolds, Jasmin Christian Blanchette, and Cesare Tinelli. Model finding for recursive functions in smt. 2015.

[32] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, volume 44, pages 37–48. ACM, 2008.

[33] Alexander Spiridonov and Sarfraz Khurshid. Automatic generation of counterexamples for ACL2 using Alloy. In *Seventh International Workshop on the ACL2 Theorem prover and its Applications (ACL2 '07)*, 2007.

[34] Rob Sumners. Checking ACL2 theorems via SAT checking. In *Third International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 '02)*, April 2002.

[35] Dale Vaillancourt, Rex L. Page, and Matthias Felleisen. ACL2 in DrScheme. In Panagiotis Manolios and Matthew Wilding, editors, *ACL2*, pages 107–116. ACM, 2006.

[36] T. Weber. Sat-based finite model generation for higher-order logic. Ph.D. thesis, Dept. of Informatics, T.U.München, 2008.