Automated Specification Analysis Using an Interactive Theorem Prover

Harsh Raju Chamarthi
Northeastern University
Email: harshrc@ccs.neu.edu

Panagiotis Manolios
Northeastern University
Email: pete@ccs.neu.edu

***Abstract*—A method for analyzing designs and their specifications is presented. The method makes essential use of an interactive theorem prover, but is fully automatic. Given a design and a specification, the method returns one of three possible answers. It can report that the design does not satisfy the specification, in which case a concrete counterexample is provided. It can report that the design does satisfy the specification, in which case a formal proof to that effect is provided. If neither of these cases hold, then a summary of the analysis is reported. The crux of our method is the use of the deductive reasoning engine of an interactive theorem prover to semantically decompose properties into subgoals that are either shown to be true or that can be *tested* to find counterexamples. Testing is interleaved with deduction in a synergistic fashion. When the deductive engine generates a subgoal that it cannot further simplify, we partially instantiate it by selecting a variable in the subgoal and assigning it a value. We then use the deductive engine to propagate the consequences of that assignment, which may lead to further deductive simplifications or to backtracking if propagation reveals a conflict. When all free variables of the subgoal have been assigned (no conflict), we have found a counterexample. We have implemented and experimentally validated the method in ACL2s, the ACL2 Sedan.**

I. Introduction

Many formal methods techniques have been developed that help designers build complex, dependable systems. At one extreme we have interactive theorem proving, which places few restrictions on the kinds of systems and properties that can be verified, but which requires well trained professionals with a deep understanding of logic and proof. At the other extreme, we have methods that find certain classes of errors in a fully automated way, but which place severe restrictions on the kinds of systems and properties they can analyze.

Is it possible to have the best of both worlds? Is it possible to have a powerful, expressive modeling language with a powerful deductive engine that can be used to interactively prove theorems *and* that can be used to automatically generate counterexamples? In this paper, we show how to do just that. We present an algorithm that makes essential use of interactive theorem proving technology but analyzes specifications in a fully automated way.

Our algorithm allows us to turn an interactive theorem prover into an *extensible*, *automatic*, analysis tool that can be used by regular engineers to provide increased assurance in the correctness of their designs. The user is responsible only for modeling and specifying the properties of their design; they are not responsible for providing proofs. It is in this regard that our approach is *automatic*. Our approach is *extensible* because it can exploit any existing or newly developed libraries of definitions, theorems and proof techniques. For example, the use of libraries for reasoning about non-linear arithmetic, set theory, the theory of lists, etc, can lead to significant improvements in our ability to prove theorems and to generate counterexamples.

The main idea of our algorithm is to use the deductive verification engine of an interactive theorem prover to semantically decompose properties into subgoals that are either shown to be true or that can be tested to find counterexamples. Deduction and testing proceed in an interleaved, synergistic fashion. When the deductive engine generates a subgoal that it cannot further simplify, we proceed to test it by selecting a variable in the subgoal and assigning it a value. We then use the deductive engine to propagate the consequences of that assignment, which may lead to further deductive simplifications or to backtracking if propagation reveals a conflict. At this level of abstraction, the process is similar to the DPLL select, assign, propagate loop. There are significant differences with DPLL, however. Variables can be over infinite domains, so selecting variables and assigning them reasonable values requires a careful analysis. Propagation in our context can involve arbitrary deductive reasoning, *e.g.*, it can prune away infinite subspaces. Backtracking also requires care because it is very difficult to analyze conflicts when variables range over infinite domains.

We present an abstract algorithm that makes minimal assumptions about the underlying theorem prover. The assumptions are outlined in Section II and the abstract algorithm is presented in Section III. We elaborate on the conrete details of our implementation in Section IV. We have implemented our algorithm in the ACL2 Sedan (ACL2s), a freely available, open-source, well-supported theorem prover that uses ACL2 as its core reasoning engine. ACL2s is an Eclipse plug-in that provides a modern integrated development environment designed to bring computer-aided reasoning to the masses. ACL2s has been used in several sections of a required freshman course at Northeastern University to teach several hundred undergraduate students how to reason about programs. We evaluate our algorithm in Section V. We present a case study on hardware verification and we also compare our algorithm with Alloy on a collection of examples from the literature.

In addition, our algorithm was used by freshmen students to debug their programs and specifications. For this purpose, the algorithm was very successful, as in almost all cases, it was able to automatically to generate counterexamples when students made mistakes. In Section VI we discuss the role of the interactive theorem prover in our method, in particular we emphasize the aspect of extensibility. Related work appears in Section VII and conclusions in Section VIII.

II. Preliminaries

In this section, we outline the assumptions our algorithm depends on. We assume that the specification language L is a multi-sorted first-order logic which can be extended by introducing new function and predicate symbols using well-founded recursive definitions, and that L is executable.

We further assume that properties (also interchangeably referred to as formulas, conjectures, or specifications) have no nested quantifiers and are of the form $hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow concl$, where hyp_i and $concl$ are formulas, and $n \geq 0$. Properties are implicitly universally quantified.

We assume the existence of an Interactive Theorem Prover (ITP) than can reason about specifications written in L. We will treat the ITP as a blackbox and all that we require from the ITP are two procedures: SMASH and SIMPLIFY.

SMASH takes as input a *goal*, a formula written in L, and returns a list of *subgoals*. We require that SMASH preserves validity, *i.e.*, the conjunction of the subgoals returned is valid iff the original goal is valid. Modern interactive theorem provers use various techniques for this, including evaluation, term rewriting, and various decision procedures for Boolean logic, linear arithmetic, and congruence closure.

SIMPLIFY takes as input an L-formula, c, and a list of assumptions (formulas), H. SIMPLIFY *simplifies* c assuming the conjunction of H, and returns a formula that is equivalent to c under H.

An *assignment* of a formula is a mapping from the free variables in the formula to values in the domain of L. An assignment may fail to satisfy all hypotheses, $hyp_1, \ldots, hyp_n$ of a formula P. In such a case, we say that the assignment is *vacuous*. Vacuous assignments are not helpful. For example, suppose that we are analyzing a compiler, whose specification says that the compiler transforms well-formed programs into semantically equivalent well-formed programs. That this property holds for ill-formed programs is trivial and not interesting. Therefore, we classify assignments as either: (1) *vacuous*, assignments that do not satisfy all of the hypotheses, (2) *counterexamples*, assignments that satisfy all the hypotheses, but not the conclusion or (3) *witnesses* assignments that satisfy all the hypotheses and also the conclusion. We note that in order to simplify the presentation, in this paper we use assumptions that are stronger than they really need to be. For example, in ACL2s, we do not require that all functions are executable.

III. The Abstract Analyze Algorithm

Analyze (Algorithm 1) takes as input a property P and a summary *summary*, which is initially empty. It analyzes P by recursively decomposing it into simpler properties, searching for counterexamples and witnesses until a stopping condition is reached. Analyze returns a status, indicating whether it reached the stopping condition, and an updated summary.

Algorithm 1 Analyze

Input: Property P, Summary *summary*
Output: Status, Summary of the analysis of P

1: **if** P is closed **then return** AnalyzeConst(P)
2: n, *status* $:= 0$, not-done
3: **while** $\neg$StopCond(*summary*) $\wedge\ n \leq$ SLIMIT **do**
4:     $A, n :=$ Search(P), $n+1$
5:     *summary* $:=$ UpdateA(*summary*, P, A)
6: **if** StopCond(*summary*) **then return** (done, *summary*)
7: $S :=$ SMASH(P)
8: *summary* $:=$ UpdateS(*summary*, P, S)
9: **if** $S \neq \{P\} \wedge S \neq \{\}$ **then**
10:     **for all** $p \in S$ **do**
11:         *status*, *summary* $:=$ Analyze(p, *summary*)
12:         **if** *status* $=$ done **then return** (done, *summary*)
13: **return** (not-done, *summary*)

Analyze first checks to see if it was given a closed property, *i.e.*, one with no free variables and does the obvious thing. Otherwise it searches for counterexamples and witnesses until either a user-defined stopping condition (StopCond) is satisfied or SLIMIT, a user-defined limit on the number of search attempts is reached (lines 3–5). The user-specified stopping condition is a predicate on *summary*, *e.g.*, our default condition is that both the number of counterexamples and witnesses found is ≥ 3. More intricate stopping conditions involving notions of coverage can also be expressed. If the user-specified stopping condition is satisfied, then we return "done" to indicate this, as well as the summary (line 6).

The procedure Search (described next) uses a DPLL-like algorithm to search for assignments that are either counterexamples or witnesses to P. To simplify the discussion, we will focus on counterexamples in the sequel, as extending the algorithms to deal with witnesses is straightforward.

Useful information is tracked in *summary*, *e.g.*, it records counterexamples (line 5), successful proofs (line 8), subgoals for which we could neither generate counterexamples nor proofs (these subgoals correspond to interesting restrictions of the original property that merit closer examination by the user), and other statistics including the number of unsuccessful search attempts, the number of counterexamples and witnesses found, the number of subgoals analyzed, etc.

If the stopping condition is not satisfied, then we semantically decompose the property P into simpler properties using SMASH (line 7). Each such simpler property is recursively analyzed (lines 10–12). In case the theorem prover is unable to simplify P, or it successfully proves P, the appropriate information is recorded (line 8) and the procedure returns with the status "not-done" and the updated summary (line 13).

Analyze terminates, as long as all of the procedures it

Algorithm 2 Search

Input: Property P with at least one free variable
Output: A counterexample (assignment) or *fail*

1: **local** Stack A of (var, val, # assigns, type, prop)
2: $A, i, x := [], 0, \mathsf{Select}(P)$
3: **while** true **do**
4: $\quad v, t := \mathsf{Assign}(x, P)$
5: $\quad P' := \mathsf{Propagate}(x, v, P)$
6: $\quad$ **if** $\neg\mathsf{Vacuous}(P')$ **then**
7: $\qquad$ **if** $t =$ "*decision*" **then** $i := i + 1$
8: $\qquad A := \text{push}((x, v, i, t, P), A)$
9: $\qquad$ **if** A is complete **then return** A
10: $\qquad i, P, x := 0, P', \mathsf{Select}(P')$
11: $\quad$ **else if** $A \neq []$ **then**
12: $\qquad$ **repeat**
13: $\qquad\quad (x, \_, i, t, P) := \text{head}(A)$
14: $\qquad\quad A := \text{pop}(A)$
15: $\qquad$ **until** $(t =$ "*decision*" $\wedge\, i \leq \text{BLIMIT}) \vee A = []$
16: $\quad$ **if** $A = [] \wedge (t =$ "*implied*" $\vee\, i > \text{BLIMIT})$ **then**
17: $\qquad$ **return** *fail*

Algorithm 3 Select

Input: Property P with at least one free variable
Output: A free variable in P

if $\exists h \in hyps(P)$ of form $x = c$ **then return** x
$G_= := \mathsf{buildEqualityDependencyChain}(P, vars(P))$
Do SCC on $G_=$, collect the leaf components in L
$leaves_= :=$ choose single x from each $l \in L$
$G_{\bowtie} := \mathsf{buildRestDependencyGraph}(P, leaves_=)$
Do SCC on $G_{\bowtie}$ to get dag $D_{\bowtie}$
if $D_{\bowtie}$ has a leaf marked constant **then**
$\quad$ **return** marked leaf
else
$\quad X :=$ the leaf with the maximum $i_=$
if X is a set (mutually-dependent variables) **then**
$\quad X_m := \{x \mid i_=(x) \text{ is maximal in } X\}$
$\quad$ **return** some vertex in X_m
else
$\quad$ **return** X

depends on terminate and as long as no property gives rise to an infinite number of calls to SMASH. Unfortunately, both of these non-terminating behaviors are possible with modern interactive theorem provers, but there are also tool-specific methods for mitigating the problem.

Searching for counterexamples

Search (Algorithm 2) takes as input a property P and searches for a counterexample by incrementally constructing a falsifying assignment to P. If it finds a counterexample, it returns it; otherwise it returns *fail*. (Recall that we also find and return witnesses, but to simplify the exposition we do not explicitly show how to do so.) The algorithm proceeds by selecting a free variable, assigning it a value and propagating this new information to obtain a partially instantiated property P'. If we can show that P' is inconsistent, we backtrack, otherwise we continue until we obtain a complete assignment.

The partial assignment is stored in stack A, which consists of five-tuples containing a variable, a value, the number of assignments made to the variable, the type of assignment (either the string "decision" or "implied"), and a property.

The main loop (lines 3–17) iteratively extends assignment A. The invariant we preserve at the beginning of the loop is that A is the current assignment, x is the free variable appearing in P that we will assign a value to next, and i, which records the number of "*decision*" assignments we have already made to x and is used to control backtracking, is not greater than BLIMIT, a natural number denoting the backtrack limit. Line 2 initializes A, i, and x so that we establish our invariant. Procedure Assign is used to assign x a value, v (line 4). Assign also returns the type of assignment. If the type is "*implied*" then assignment A and property P imply that x has to have value v. For example, suppose that $x = c$, where c is a constant expression is a hypothesis of P. Then v will be $[\![c]\!]$ (what c evaluates to) and this is an *implied* assignment. This is but one of the optimizations Assign performs. If the value of x cannot be uniquely determined, then Assign assigns it a value as outlined in the next section and returns "*decision*" as the type of the assignment. Next, in line 5, we use the theorem prover to propagate the information that x has value v in P, obtaining property P' (Propagate is described in a later section). P' is vacuous if it contains falseas a hypothesis.[1]

There are now two options. First, perhaps P' is not vacuous. In this case we increment i iff we made a decision. We then push the appropriate tuple on A, which includes the selected variable x, the value it was assigned v, i, the type of assignment, and the property the previous values depended on, P. We then check if A is complete; if so, we have a counterexample and we return it. If not, then P' has at least one free variable, so we select a new variable, re-establish the previously mentioned invariant, and iterate. Second, perhaps P' is vacuous, *i.e.*, we discovered a conflict. In this case, if A is not-empty, we backtrack, by popping A, until we go past a decision whose backtrack limit has not been reached or until $A = []$. After backtracking, we can only continue with the while loop if we did not exceed our backtracking limit. If we did exceed the limit, then the test in line 16 passes and we return *fail*. Notice that if the test passes, then P' was vacuous and every decision on A exceeded the backtrack limit.

Selecting variables to assign

Select (Algorithm 3) takes as input a property P with at least one free variable and returns a variable occuring free in P. The order in which variables are selected is very important. The idea is to select unconstrained variables (this

[1] Note that $hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow concl$ is equivalent to $hyp_1 \wedge \cdots \wedge hyp_n \wedge \neg concl \Rightarrow$ false. In the above exposition, this implies that our vacuity check on P' also succeeds if $concl$ is true.

minimizes the probability that we assign the variable a value that is inconsistent with the current assignment) on which other variables depend (so that other variables will be "implied"). Consider the following motivational example, where x, y, z, and w are constrained to be integers and h is a function from integers to integers.

$$(P)\ z = yv^3 \wedge y = h(x) \wedge w = h(y) \wedge v = h(6) \Rightarrow z > w^2$$

Since we are interested in finding counterexamples, recall that we have four constraints to satisfy: the three hypotheses and the negated conclusion.

Which variable should we select and assign a value to first? Notice that v is equal to a constant expression, so the value of v is implied. We select such variables first, as per line 1. Assign will assign v the value $[\![c]\!]$; we will discuss Assign in a later section.

Notice that equality constraint fixes the value of y as soon as x is assigned, and the value of z and w as soon as y is assigned a value that does not falsify other constraints. Clearly, choosing x before choosing y is beneficial from the point of view of computation *i.e.*, we just evaluate $h(x)$ to obtain the value of y. Selecting y before x, causes complication in satisfying the constraint $y = h(x)$, since computing the inverse of a number-theoretic function might be non-trivial. Moreover, any constraint solver used in Assign might not be powerful enough to handle the complex arithmetic of h. Treating *equality* in a special manner we can see that there is a certain relation among the variables of the constraints that is similar to the notion of data dependency in compiler literature. The idea behind the algorithm is to select the variable with the *least dependency*, breaking down the task of simultaneously solving the constraints, into a more local directed approach of solving the constraints one by one; we want to finally select variables in an order such that we can reduce the chances of running into an inconsistency (vacuous assignment) and backtracking. The procedure Select (Algorithm refsec:selectalg) first canonicalizes the equality constraints in input P. We basically make two passes over P constructing directed graphs first characterizing the equality dependency relation among variables and then taking care of the rest of the dependencies. First we construct an *equality dependency graph* $G_=$ for P, that initially consists of no edges, with all the free variable in P as nodes. The graph is constructed by iterating over the constraints of P using the following three rules, implemented in procedure buildEqualityDependencyChain (line 2) and shown after this note.

Note: A leaf node (variable) is a node with no outgoing edges and no dependency (of interest) on other variables. The goal of our algorithm is to pick and return such a variable. We will sometimes refer to them as independent variables. We assume 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) Case: $x = c$, where c is a constant expression. Mark x to be a constant leaf node (no outgoing edges). Note: Since the constraints are canonicalized, we dont have to check for $c = x$.
2) Case: $x = y$. Add bidirectional edge between node x and node y.
3) If P has a constraint of the form $x =$ *fterm* such that $y \in$ freeVars(*fterm*) and $x \notin$ freeVars(*fterm*), we add a directed edge from node x to node y, unless x is a constant leaf node.

After $G_=$ is constructed, its strongly-connected components are computed. A single representative variable is picked at random from each leaf component and stored in $leaves_=$ (lines 3–4). Using only the $leaves_=$ as initial nodes, and no edges, we make a second pass through all the constraints in P building a *non-equality dependency graph* $G_{\bowtie}$. The following rules form the core of buildRestDependencyGraph. If more than one rule applies, then the rule that comes earliest in the following sequence overrides the others.

1) If P has a constraint of the form $x \bowtie y$ where $\bowtie \in \{<, \leq, >, \geq\}$ we don't draw an edge. *Intuition*: Constraint $x > 3$ is as easy to satisfy as $42 > y$, so we avoid complicating the graph.
2) If P has a constraint of the form $x \bowtie$ *fterm* such that $\bowtie$ is a binary relation, $y \in$ freeVars(*fterm*) and $x \notin$ freeVars(*fterm*), we add an $\bowtie$-edge from node x to node 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.
3) If P has a constraint of the form *R(term*$_1$, *term*$_2$, ..., *term*$_n$*)*, such that $x \in$ freeVars(*term*$_i$), $y \in$ freeVars(*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,

After the graph is constructed, using the aforementioned rules, its dag $D_{\bowtie}$ is obtained by SCC analysis. If the dag has a leaf node marked *constant*, we return that variable. Otherwise the procedure picks the leaf component, with the maximum number of nodes that can reach it in $G_=$, for any node (component) x, we denote this by $i_=(x)$. So we pick the component that has the potential to force the maximum number implied assignments via the equality constraint (lines 7–10). If the component has just one node, then usually it is an independent variable, in which case we return it. If there are more than one variables to choose from (in case of multiple variables in the connected component), the procedure returns the variable with maximum $i_=$ value, if there is a tie, we simply pick a variable randomly (lines 11–13). Note that Select tries to ensure the following rule of thumb: select a variable only when every variable it *depends* on has already been assigned a value; this is not always the case.

Algorithm 4 Propagate

Input: Var x, Value v, Property P
Output: Property obtained by propagating $x = v$
1: $hyps := x = v \wedge \mathsf{hyps}(P)$
2: $shyps := \mathsf{simplifyAssumingRest}(hyps)$
3: $sconcl := \textsc{Simplify}(\mathsf{conclusion}(P), shyps)$
4: **return** $\bigwedge shyps \Rightarrow sconcl$

Propagating new assignments

After a variable is assigned a concrete value, we use the theorem prover to propagate this information, with Propagate, shown in Algorithm 4. This procedure takes as input a variable x, the value v assigned to x, and a property P. It adds the constraint $x = v$ to the list of hypotheses of P, calls simplifyAssumingRest, a procedure that given a list of hypotheses calls SIMPLIFY to simplify each hypothesis as much as possible, under the assumption that the rest of the hypotheses are true. The resulting simplified hypotheses are stored in *shyps* (line 2). Similarly, the conclusion is simplified, assuming all the formulas in the list $shyps$ are true (line 3). The resulting propery is returned.

A. Example

We illustrate the working of Search on a simple example involving numbers and some arithmetic functions. Consider the following property P defined on integers x, y, z, w; *hash* and min are textbook *hash* and *minimum* functions.

$$x = hash(y) \wedge y = hash(z) \wedge z > 0 \wedge w < min(x, y) \Rightarrow w < z$$

Before the main search loop begins, a variable is selected to be instantiated. The dependency graphs for P (constructed following the aforementioned rules) are shown in Figure 1 and 2.

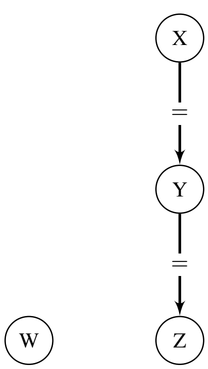

Fig. 1. $G_=$ for P

Fig. 2. $G_{\bowtie}$ for P

Notice that only the leaves of $G_=$ appear as nodes in $G_{\bowtie}$. Both w and z are leaves in $D_{\bowtie}$ (same as $G_{\bowtie}$) We pick z since its $i_=$ value, which is 2 (both y and x implied as soon as z is decided), is greater than w's $i_=$ value, 0. Since z is not a set (component) we simply return z. After having selected the variable to instantiate (z), we use Assign to pick a value for it, satisfying the local constraint on it, $z > 0$, along with the implicit constraint that z is an integer. Lets say the oracle procedure Assign picked 34. Then we propagate this assignment by adding the constraint $z = 34$ in P and using the ITP to simplify the hypotheses and conclusion in light of this new information. Propagate returns the following simplified property:

$$P' : \quad x = 3623878690 \wedge y = 268959709 \wedge w < min(x, y) \Rightarrow w < 34$$

whose dependency graphs are shown in Figure 3 and 4.

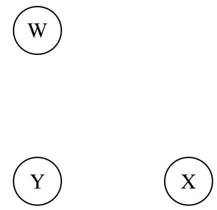

Fig. 3. $G_=$ for P'

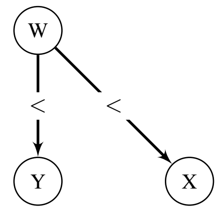

Fig. 4. $G_{\bowtie}$ for P'

Since false does not appear in the hypotheses(and neither does true in the conclusion), P' is not inconsistent and we add $z = 34$ to the partial assignment A and the search for the rest of the assignment is continued.

The motivation for Propagate is that one assignment to a variable, should result in assignment of the maximum number of remaining variables. In this case, the assignment to z, results in y and x being fixed to constants. Since both x and y are leaf nodes in figure 4 and have the same $i_=$ value, we will randomly choose one (the order does not matter). Lets say y is selected. Search directly assigns it a value by virtue of the equality constraint $y = 268959709$. Notice that this is an assignment of type "implied" and was propagated due to the decision assignment ($z = 34$) by the oracle procedure Assign in the previous iteration. This information is again propagated resulting in the further grounded property:

$$P'' : \quad x = 3623878690 \wedge w < min(x, 268959709) \Rightarrow w < 34$$

whose dependency graphs is shown in Figure 5 and 6.

Fig. 5. $G_=$ for P''

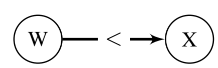

Fig. 6. $G_{\bowtie}$ for P''

x is clearly the lone leaf node in Fig 6, it is selected and directly assigned using the equality implication $x = 3623878690$. This assignment is further propagated, resulting in the almost grounded property having just one free variable:

$$P''' : \quad w < 268959709 \Rightarrow w < 34$$

Assigning w (using implicit constraint that w is an integer and the local constraints $w < 268959709$ and $w \geq 34$) a value 33 or value 268959710, will lead to inconsistency (after the propagation), in which case we need to throw away the current assign and decide a new value for w. If in the process we exhaust the limit on number of assigns (BLIMIT) for w we backtrack all the way to the decision variable z, by undoing the implied assignments to x and y, in A, popping P'' and P' from S and continuing (the main search loop). If an assign to w, say $w := 42$, did not lead to an inconsistency, then we have a complete assignment A, we quit the loop and return A, which is a counterexample of P.

IV. Concrete Algorithm in ACL2s

We have implemented the proposed method in ACL2 Sedan (ACL2s) [11]. We employ the ACL2 interactive theorem proving system [19] to provide the interface methods SIMPLIFY and SMASH. The specification language, also called ACL2, is *untyped*[2]. To attract the common programmer (or designer) to use our tool, we provide a data definition facility (`defdata`) in ACL2s to specify various kinds of type idioms, like record types, enumeration types, union types etc, commonly found in most modern programming languages. The engineering of the interface with the ACL2 theorem prover and the extension to ACL2, in support of this interface, the data definition facility, and other ACL2-specific details are described in [7]. The Analyze algorithm is simulated using the computed hints mechanism of ACL2 (see [7]). The implementation of the Search, Select and Propagate closely follow their abstract algorithms shown in the previous section. We will briefly describe the implementation details of the Assign method that had been left unspecified.

In view of delegating most of the heavy work to the theorem prover we incorporated the lightweight method of random testing inspired by the success of Quickcheck-like tools [8]. Alternatively we could also have chosen more heavyweight constraint solving techniques (*e.g.*, SAT/SMT Solvers). ACL2 formulas tend to be executable, hence testing in ACL2 simply involves executing a formula under an instantiation of its free variables. To assign a value to a variable, we need to know its domain, which in a given formula is determined by the "type-like" hypotheses constraining the variable. The domain can be characterized by an *enumerator* which is a surjective function from natural numbers to elements of the domain.

Thus the problem of supporting user-defined data definitions and automatic testing (sampling) is elegantly solved by adding a notion of an enumerable type to our untyped specification language (ACL2). This is accomplished in our tool, using the *defdata* form that introduces a "type", by virtue of a predicate and an enumerator being automatically generated. Such predicates are used to specify the type-like constraints for a variable in a property. In addition to types introduced by *defdata* and the primitive types, all elements of the ACL2 value universe are treated as singleton types, and a special type *All* denotes the universal type. The *defdata* form additionally takes care of maintaining a type hierarchy called (*datadef ordering graph*) which captures the subtype relationship among the introduced and primitive types.

[2]In fact many specification languages are untyped, an interesting discussion can be found in an article by Lamport and Paulson [21].

Algorithm 5 Assigning free variables

Input: Property P, free variable x
Output:
$typ :=$ inferType(x, P)
if typ denotes a constant expression **then**
  $mtyp := typ$
  return $[|e|]$, "$implied$"
else
  $mtyp :=$ refineType(typ, P)
$e :=$ BuildValueExpr$(mtyp, nil, 0)$
if $mtyp$ is a singleton type **then**
  return $[|e|]$, "$implied$"
else
  return $[|e|]$, "$decision$"

Separation of concerns between enumerators and random number generators also gives us the flexibility to choose between random sampling and bounded exhaustive sampling of test data. Assign does static analysis to infer the (enumerable) type of a variable from the type hypotheses of P, if the domain of the type is greater than one, we *decide* a value to return (using the enumerator and the chosen sampling distribution), otherwise, we simply return the *implied* singleton value. Assign is shown in Algorithm 5, it takes a variable x and a property P, uses InferType to extract the "type" typ of x. InferType uses straightforward syntactic analysis of the type-like hypotheses constraining x in P. If typ denotes a constant expression, we simply return it, otherwise, Assign refines typ using procedure refineType as much as possible to find minimal type information, $mtyp$. Finally the minimal type expression is used to build a value expression e using the procedure BuildValueExpr. The procedure finally returns the value of e, along with the information about the type of assignment. Note that e is of type $mtyp$. If the domain of $mtyp$ consists of exactly one value object, then clearly, there is no choice but to return it as the value to be assigned, such an assignment is named "implied". Otherwise, if the domain of $mtyp$ has more than one value object, then we decide a value object to return, and name the assignment, "decision". The choice of which value object to return is hidden in the implementation of BuildValueExpr, and is determined by the user-specified sampling distribution (by default random sampling is used), it builds a value expression whose evaluation performs the actual sampling of the domain of $mtyp$ (*e.g.*, for $mtyp$=integer, the value expression '(nth-integer 42) might be built).

Currently refineType is a no-op in our implementation. But we believe it is important to obtain the minimal possible type information from the conjecture, since smaller the domain of the variables to be instantiated, the higher the probability of hitting counterexamples (and witnesses). We present our design of the refineType algorithm below.

Given variable x appearing in property (conjecture) C, we want to determine the (minimal) type of x. The type has to be an element (or union of elements) of our type graph G_T (*datadef ordering graph*). Recall that G_T can be turned into a dag by performing a strongly connected component analysis, so, wlog, we assume that G_T is a dag. Formally, we want to compute the following:

$$\mathit{Type}(x) = \{t \ : \ t \in G_T, P(C, x, t), \\ \forall t' \in T :: t' \subset t \Rightarrow \neg P(C, x, t')\}$$

i.e., we want to compute the set of all types t in our graph G_T such that under the hypothesis in property C, x is provably always an element of t (denoted by $P(C, x, t)$ above) and there is no proper subtype, t' of t such that x is provably always an element of t'.

There are several problems to address. First of all determining $P(C, x, t)$ precisely is an undecidable problem, so we will instead use $P(C, x, t)$ to denote that given the hypotheses in conjecture C, *our theorem prover can prove* that x is always an element of t.

It would be nice if $|\mathit{Type}(x)| = 1$, but unfortunately, it is possible for there to be more than one element in $\mathit{Type}(x)$, *e.g.*, suppose that the hypotheses in c state that x is a positive even integer, and that T includes four types: the set of integers, the set of rationals, the set of positive rationals, and the universe. Then, $\mathit{Type}(x)$ contains two elements (the set of integers and the set of positive rationals).

We now consider algorithms for computing *Type*. Here is a first attempt.

Algorithm A:

1) Find t_0 such that $P(C, x, t_0)$. We do this with a simple static check and we can always just set t_0 to be All, the universe.
2) Traverse G_T collecting all types t such that $P(C, x, t)$ but for all t', where t' is a successor of t, we have $\neg P(C, x, t')$. We can use depth-first search to do this in linear time.

Algorithm A is incorrect because our theorem prover is not monotonic, *i.e.*, it may prove h but not g even if $h \Rightarrow g$. For example, suppose that G_T contains the following edges $t_1 \Rightarrow t_2$, $t_2 \Rightarrow t_3$, $t_3 \Rightarrow t_4$. It is possible that the following hold: $P(C, x, t_1)$, $\neg P(C, x, t_2)$, and $P(C, x, t_3)$. According to our definition of $\mathit{Type}(x)$, we should return $\{t_3\}$, but Algorithm A will return $\{t_1\}$. We could just query the theorem prover for every type in G_T, to $P(C, x, t)$ for all $t \in T$, but this seems wasteful since calls to the theorem prover can be expensive. Algorithm 6, below shows how to do better by using counterexample generation (but with simple random testing using the type information from InferType to avoid mutual-recursion). Each $t \in T$ will have a label associated with it which is either "?" (indicating that we do not know if x is always in t), or "yes" (indicating that x is always in t), or "no" (indicating that x is not always in t). When we initialize G_T, we label all nodes with "?", except All, which is labeled with "yes". When we label t "yes", we also label nodes that can reach t with "yes". Similarly, when we label t with "no" we label all nodes that t can reach with "no". We

Algorithm 6 Refine Type

Input: Initial type t_0, Constraint C, Variable x
Output:

local Stack M (of Minimal types to be returned)
Initialize G_T
Label t_0 with "yes"
$t := t_0$
while DFS on G_T, current visited node $= t$ **do**
    if $\mathit{label}(t)! =$ "?" **then**
        $P := \mathit{hyps}(C) \Rightarrow x$ satisfies predicate of type t
        if random instantiation of P returns false **then**
            setLabel(t, "*no*", G_T)
        else if SMASH$(P) =$ true **then**
            setLabel(t, "*yes*", G_T)
        else
            skip
$t, M := t_0, \mathit{empty}$
while DFS on G_T, current visited node $= t$ **do**
    if $\mathit{label}(t) =$ "*yes*" **then**
        if $s \in$ Successors$(t) \Rightarrow \mathit{label}(s)! =$ "*yes*" **then**
            push(t, M)
return union of types in M

maintain the invariant that if a node is label "yes" then so are all of the nodes that can reach it and that if a node is labeled "no" then so are all nodes it can reach. This means that when we propagate labels, we stop as soon as we find a node with a label that differs from "?".

Algorithm 6 requires a linear number of queries to the theorem prover and runs in linear time (in the size of G_T). Also, since in most cases we do expect $|\mathit{Type}(x)| = 1$, it is much more efficient than the algorithm that queries the theorem prover for every type in G_T.

V. Experimental Evaluation and Discussion

We present two experiments[3] to evaluate our method. In Section V-A, we present an in-depth hardware case-study, analyzing the design of a simple, yet non-trivial, pipelined machine, demonstrating the effectiveness of our method in uncovering subtle design errors. In Section V-B we compare our method with the popular Alloy method (Alloy modeling language and Alloy Analyzer). We modeled various Alloy examples in ACL2 and analyzed them with our method. We find counterexamples to all failed properties (falsified by Alloy), but more importantly we prove all the properties that Alloy posits are theorems (based on the absence of small counterexamples). Surprisingly, in addition to the counterexamples, we also found all the proofs, automatically.

A. Hardware: Finding hazards in a Pipeline Machine

Pipelining is a key optimization technique used to increase performance in modern microprocessors. The *instruction-set*

[3]We recommend the reader download the experiments from http://ccs.neu.edu/home/harshrc/fmcad11

architecture (ISA) model is a natural functional specification for any pipelined design. The correctness of the implementation *i.e.*, *machine architecture* (MA) can be established by showing that all behaviors (execution traces) of MA are observationally equivalent to behaviors of its specification (ISA).

We analyze a three stage pipeline, consisting of fetch, read, and execute/write-back stages. The machine is based on previous work [23]. The machine fetches an instruction pointed to by the program counter in the fetch stage, reads the source register from the register file in the read stage, and updates the destination register with the result of the operation it performs (execution) in the write-back stage. The primary challenge in designing a correct pipeline implementation is respecting program dependency and avoiding resource conflicts among instructions that are in different stages of the pipeline. Consider the following sequence of ADD instructions:

$$I_1 : r_3 = r_2 + r_1 \; ; \; I_2 : r_4 = r_3 + r_2$$

Instruction I_2 will read stale data for register r_3, if read phase of I_2 overlaps with the execution phase (write-back) of instruction I_1. In such a scenario (called Read-after-Write data hazard), to correctly handle the data dependency, the pipeline must be *stalled* to allow the older instruction (I_1) to execute and update the destination register (r_3) before the younger dependent instruction (I_2) reads it. In our pipeline machine model, we will on purpose introduce a design error by failing to stall the read for I_2 in the above scenario. Another scenario that we consider is related to handling of branch/jump instructions. By the time, the program counter is updated to fetch from the target of a BEZ/JMP instruction, subsequent instructions from the sequential program code have already been fetched. To prevent the wrongly fetched instruction from polluting the architectural state (control hazard), it is required to invalidate the latches holding information related to instructions from the wrong execution path. A common error occurring in initial phases of the design of a pipeline machine, is to forget invalidating latch 2, in the scenario that latch 1 is invalid.

The objective of the experiment was to evaluate the effectiveness of our method to find these important and subtle design errors (data and control hazards). How do we find these bugs using our method? Given that the designer has written both the ISA and MA models of the pipeline machine, one just needs to formalize the aforementioned correctness definition and analyze it. We will use a notion of refinement, where the main idea is to show that infinite behavior of MA and ISA are observationally equivalent under an appropriate refinement map. By using the theory of Well-founded equivalence bisimulation (WEB) refinement, we can establish this by proving a local property that only requires reasoning about MA states, their successors, and ISA state and their successors [22]. The refinement map is straightforward, except for the matter of relating the program counters of MA and ISA states. Since the observable effect of any instruction only appears in the write-back stage, the observable program counter is simply the PC value of the oldest instruction in the pipeline. Let M' denote the state of the machine after it has taken one step *i.e.*, it has been run for one hardware clock cycle. Then the safety part of our WEB refinement proof obligation is that if ISA state S and MA state M are observationally equivalent, and both take a step to S' and M' respectively, then either S is observationally equivalent to M', or S' is observationally equivalent to M' (stepping MA for one cycle resulted in an observable architectural-fallback change).

Analyzing this high-level property, our method is able to uncover both the design errors in our MA machine which manifested as hazards (in under 2 minutes). The counterexamples (instances of MA that falsified the safety property) were illuminating; they pointed out the kind of hazards and the scenarios in which they occurred. We recommend the reader to play around with the model provided to see if the tool can uncover other scenarios he/she has seen before.

A few observations are in line. No assertions were provided. No lemmas were written down. No manual tests (microprograms) were provided as inputs. No test driver needed to be given. The only effort on part of the designer was in writing the ISA and MA models in ACL2, defining the datatypes (used for automatic test data generation), specifying the abstraction function (for observational equivalence) and formulating the high-level correctness property.

B. Software: 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 off-the-shelf SAT solvers 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. However to enable feasible automatic analysis, it has poor support for two features that we feel naturally apply in many types of modeling/design examples: recursive definitions and arithmetic. The ACL2 language, on the other hand, has excellent support for recursive definitions (in fact, in ACL2, most interesting properties are expressed using recursive definitions) and arithmetic [27]. In view of this (and our limited Alloy expertise), we avoid doing a comparison on problems that we perform well (*e.g.*, the property involving `hash` function in Section III is inexpressible in Alloy due to absence of multiplication), and restrict ourselves to examples (from the Alloy distribution) that we think Alloy performs well on.

We analyzed 12 properties from 4 Alloy problems (specifications), except the markSweep problem, all the others are from the Alloy book [17] and can alternatively be downloaded from the Alloy distribution.[4] Table 1 shows results, comparing

[4] Alloy Analyzer can be downloaded from http://alloy.mit.edu/alloy4

Property	Alloy Analyzer			Our method	
	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
markSweepSound	8	47.34	–	0.28	QED
markSweepComplete	7	58.12	–	0.34	QED

TABLE I
COMPARISON WITH ALLOY ANALYZER (AA)

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 scope for which AA can check the property before exceeding the 2 minute time limit, or the 1 GB memory limit. 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. The last 2 rows are the Soundness and Completeness properties of the mark-and-sweep model, where live (reachable from root) nodes of the heap are *marked* and garbage (unreachable from root) nodes are *sweeped* into a freelist. The mark-and-sweep Alloy model was taken from an experiment in [14] where Alloy specifications are automatically translated to SMT2 language supported by the Z3 SMT solver [10]. We would have also liked to provide experimental comparision with the Alloy-to-SMT approach taken in [14], but we did not have access to their implementation.

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 the rich datatype support provided by ACL2s [11]. These respectively make use of the ordered sets library and the records library [24], [20], [9] 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. No intermediate lemmas were provided, no hint or guidance was offered to the theorem prover, the proof of *pasteAffectsHidden* by ACL2s was as unassisted as the counterexample generated by Alloy for *cutPaste*. The counterexamples generated by our method, in few cases, required some manual assistance when random testing (default) was not good enough to catch the counterexample, we had to bound the types, to simulate automated bounded testing. But this is not hard to automate and is a shortcoming in our implementation rather than the method itself. If you are curious how the set and map theory libraries helped in the automated proofs, one can look at the proofs obtained in ACL2s. For example, the proof of *pasteAffectsHidden* succeeded primarily by the use of four rewrite rules (enabled by inclusion of set and record libraries). The first rewrite rule says, that union operation of sets is symmetric, the other rewrite rules are the classic record update axioms [25], where r is a map (record) and a,v stand for addresses and values respectively.

```
1. (equal (union x y) (union y x))
2. (equal (mget a (mset a v r)) v)
3. (implies (not (equal a b))
            (equal (mget a (mset b v r))
                   (mget a r)))
4. (implies (not (equal a b))
            (equal (mset b y (mset a x r))
                   (mset a x (mset b y r))))
```

In experiments shown in [14], it is found that the correctness of the translated (from Alloy into Z3) mark-and-sweep model could not be proven by Z3; the authors mention that this problem is particularly difficult due to the fact that the simulation of recursion involved in mark-and-sweep by transitive closure results in deeply-nested quantifiers that Z3 cannot handle. We modeled the problem in ACL2, used sets and maps as mentioned before, the mark procedure (involving transitive closure) is modeled using a simple recursive definition. We then formalize the following properties that imply correctness:
Soundness: *No live node appears in the freelist*
Completeness: *All garbage nodes are eventually collected*
We were able to prove the above properties automatically. Again, no domain-specific lemmas were used, no hints were given to the theorem prover, no expert knowledge of theorem prover was required. This might seem surprising, and we must deflate some optimism here, by pointing out that this automation will not scale for non-trivial models, but surely we must not overlook the effectiveness of powerful libraries (*e.g.*, set reasoning) by the tool-writer put to use by the choice of right abstractions (*e.g.*, using set datatypes) by the designer.

VI. DISCUSSION

There are various ways in which the ITP helps the search procedure, it might reduce the complexity of the constraints (characterizing the solution space of a property), pruning away whole (possibly infinite) subspaces from the complete search space, it might decompose the constraints in a manner that focuses the search on interesting portions of the search space, it might help refine the type information, furthur reducing the search (sample) space for each variable, and finally it might massage the constraints to an equivalent form more amenable to the heuristics and internals of the search algorithm. Thus, the more powerful the theorem prover, the more powerful the search for counterexamples. Unlike other tools, an ITP can be customized, it can be made more powerful, by proving lemmas, an user can program the ITP's main deductive reasoning engine (rewriter), enabling it to simplify more formulas than was possible before. It is in this regard that we call our tool *extensible*. Adding a domain-specific theory (library) of rewrite rules transforms the ITP into a powerful reasoning engine for that domain. For example, the standard arithmetic library enables ACL2 to simplify even non-linear arithmetic terms, something that is beyond more automatic tools. So even though, this does not mean that our tool, by virtue of including domain-specific libraries will automatically, answer an *yes* (valid) or *no* (invalid) when posed with a conjecture from that domain, the simplification of the conjecture by the ITP rewriter, decreases the probability of answering neither *yes* or *no*. Thus one can view an ITP as a very powerful preprocessor, which can be used to simplify the problem as much as possible, before resorting to decision procedures (like SAT and SMT Solvers).

But what if for a particular property, the tool neither produces a proof, nor a counterxample? There are two main issues at hand here.

Firstly, how best can we summarise this result? Can we give some sort of a coverage result? Say our method decomposed the property being checked into a 100 subgoals, and the ITP proves 99 subgoals and fails to prove one. How do we translate this information into the more traditional coverage metrics that engineers are used to, for *e.g.*, branch coverage. It seems the coverage notion obtained from a theorem prover is stronger than traditional software coverage notions.

Secondly, what should the user do now. There are three choices for the user. First, the user might run the tool with different parameters for the search algorithm, for example, our search algorithm uses a random test-case generation method to *assign* variables, one can tune the distribution of data, choose a bounded-exhaustive test strategy and so on. We have not yet implemented coverage metrics, but we plan to, once its done, the user can run the tool till it meets the coverage criteria given by the user. Secondly, the user might add a third-party library that provides reasoning power for his domain of interest. Finally we note that if the user is willing, he might directly help the tool by providing hints that will increase the reasoning power of the ITP (and in turn the counterexample search). This fact is worth emphasizing, since often the engineer who has designed a system, has a fairly intimate knowledge of the main characteristics of the system, and might be certain of some facts, and uncertain of others. For example, he might have a hunch, about what scenario might reveal a bug *i.e.*, the engineer has some insight that might help prune the search space (for counterexamples). As an illustration here is an anecdote of how Euler disproved one of Fermat's conjecture that is recounted in the book [12]. Fermat had conjectured that all numbers of the form $2^{2^n}+1$ are primes, and showed this to be true for $n = 1, 2, 3, 4$. But $2^{2^5}+1 = 4294967297$ is a huge number, and in those days of hand calculations, it would have been a painful task to enumerate factors of 4294967297. Studying these number, Euler used some mathematical insight to break down the problem, he found that all factors of $2^{2^n}+1$ should be of form $k * 2^{n+1} + 1$ for some k. Using this insight, he greatly narrowed down the list of potential factors to be considered and in fact it turned out, he didnt have to put in much effort, he found a counterxample to Fermats conjecture for the very next number $n = 5$, the factor 641 was found for $k = 10$. Thus an engineer might be reluctant to undertake a full formal proof, but might be willing to formalize the facts (insights) he thinks are obvious as lemmas (rewrite rules), and if the ITP proves them, then it results in more powerful tool both in terms of refuting (searching for counterexamples) and proving the top-level main conjecture, since the ITP uses these simple facts and possibly simplifies away some complex constraint that was blocking the search for the counterexample, or was hindering the top-level proof. Thus our method provides a user-customized migration path from testing to full proof.

VII. Related Work

Counterexample Generation in Interactive Theorem Provers

Random Testing is a well-studied, scalable, lightweight technique for finding counterexamples to executable formulas. Many Interactive Theorem Provers motivated by the success of QuickCheck and related random testing tools [8] have implemented random testing libraries *e.g.*, Isabelle/HOL [1], Agda [13] and PVS [26]. 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 logic with limited expressiveness. The major constraint on such approaches is that a counterexample to the translated formula should also be a counterexample to the original formula. However, the absence of a counterexample does not imply that the conjecture is true. Some tools making use of the above technique are Pythia [28], SAT Checking [29], Refute [30] and Nitpick [2]. The work mentioned above has the same goal as our work: automatically exhibit counterexamples to false properties. However, unlike our work, none of the above mentioned approaches *use* the interactive theorem prover to generate counterexamples for arbitrary properties.

Combining Testing and Interactive Theorem Proving

Ideas for using formal specifications and combining simplification (theorem-proving) and testing date back to at least 1981 [6]. One of the first examples of combining testing and interactive theorem proving was carried using Agda [13]. Random testing was used to check for counterexamples, and the point was made that the user could apply random testing also to subgoals. Another instance of leveraging a theorem prover to improve testing is the HOL-Testgen tool [3] which was designed for specification-based testcase generation. Compared to the above approaches, our method has a more fine-grained and tighter integration with the interactive theorem prover.

Automatic Analysis tools

Alloy is a declarative specification language based on relations and sets. The Alloy Analyzer can automatically find small counterexamples to Alloy specifications. This is done by translating the Alloy specification into a boolean satisfiability formula and using an off-shelf SAT Solver to find a solution (model). In contrast, we primarily make use the deduction power of an ITP to simplify the problem at the specification level and then using a search algorithm which uses both the ITP and testing. As a result, we can, in addition to finding (short) counterexamples, 1) prove their non-existence and 2) find deep counterexamples that the bounded method of Alloy will miss. Also see V-B for comparision of our implemented method (ACL2s) and Alloy. We believe some of the techniques are complementary, and both can benefit from each other.

A. Dynamic Test Generation

There has been much recent work on using symbolic execution for dynamic test generation [15], [5], [4], [16]. Since such tools differ significantly from our method, we will briefly mention the conceptual similarities and then make a case for how these tools can benefit from our work. These tools are similar to our method in the sense, that when we decompose a property (say model P and assertion A), one can think of it as symbolic execution, but interleaved with simplification driven by the deduction engine (ITP). DART [15] and SAGE [16] concretely execute a given program P starting from a random (or some well-formed interesting) input and collect symbolic path constraints (symbolic execution) on the side. EXE [5] and KLEE [4] perform mixed symbolic and concrete execution (user can manually set some inputs as concrete, leaving others symbolic). At interesting program points, say assertions, the symbolic path constraint and the negated assertion ($\neg A$) are given to a constraint solver, a solution obtained is a counterexample. When the constraint is too complex for the constraint-solver to handle, some tools (DART) randomly pick certain input variables to be replaced by their concrete values, perhaps simplifying the constraint within the reach of the constraint-solver. Consider our example from III-A, unless they exactly pick z (found by analyzing the variable dependency graph), the aforementioned tools will fail to handle it. But these tools dont have a systematic procedure of choosing which variables to be replaced. We believe our Select algorithm can provide a simple starting point for such a procedure.

VIII. Conclusions

We presented an algorithm that uses an interactive theorem prover to automatically analyze models and specifications. Our approach has several advantages over related work. It allows designers to use expressive languages to model systems at various levels of abstraction, with support for data structures, arithmetic, and recursive procedures. It is fully automated and compares favorably to existing methods for analyzing high-level models. Our algorithm is implemented and freely available in ACL2s, the ACL2 Sedan.

IX. Acknowledgments

We would like to thank Peter Dillinger, Mitesh Jain, and Matt Kaufmann for their invaluable help. This research was funded in part by NASA Cooperative Agreement NNX08AE37A. This article reports on work supported by the Defense Advanced Research Projects Agency under Air Force Research Laboratory (AFRL/Rome) Cooperative Agreement No. FA8750-10-2-0233. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *SEFM*, pages 230–239. IEEE Computer Society, 2004.

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

[3] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *FATES*, volume 3395 of *LNCS*, pages 16–32. Springer, 2004.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224. USENIX Association, 2008.

[5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[6] R. Cartwright. Formal program testing. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 125–132, New York, NY, USA, 1981. ACM.

[7] H. R. Chamarthi, P. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. In *Ninth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '11)*, November 2011.

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

[9] J. 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.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[11] P. Dillinger and P. Manolios. ACL2 Sedan homepage. See URL `http://www.acl2s.ccs.neu.edu/doc`.

[12] W. Dunham. *Journey through genius: the great theorems of mathematics*. Wiley, 1990.

[13] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In D. A. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 188–203. Springer, 2003.

[14] A. A. E. Ghazi and M. Taghdiri. Relational reasoning via smt solving. In *FM*, 2011.

[15] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[16] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*. Citeseer, 2008.

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

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

[19] M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://-www.cs.utexas.edu/users/moore/acl2`.

[20] M. Kaufmann and R. 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.

[21] L. Lamport and L. C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21:502–526, May 1999.

[22] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001.

[23] P. Manolios. Refinement and theorem proving. In M. Bernardo and A. Cimatti, editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems: Hardware Verification*, volume 3965 of *LNCS*, pages 176–210. Springer-Verlag, 2006.

[24] P. Manolios and M. Kaufmann. Adding a total order to ACL2. In M. Kaufmann and J. S. Moore, editors, *Proceedings of the ACL2 Workshop 2002*, 2002.

[25] J. McCarthy. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, 19, 1967.

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

[27] D. Russinoff. A mechanically checked proof of ieee compliance of the floating point multiplication, division and square root algorithms of the amd-k7 processor. *LMS Journal of Computation and Mathematics*, 1(-1):148–200, 1998.

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

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

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